(cTOASTER) `cupcake` Configuration and Build System

Ian Ross, with contributions from:
Andy Ridgwell

February 7, 2024

This document[1] describes the new configuration and build system developed for the `cupcake` release of the **cTOASTER** model. Documentation is divided into three sections, one for users of the model, one for those who wish to modify the model and one for those concerned with maintaining and extending the configuration and build system infrastructure.

1 For users

1.1 Quick start

Open a shell in your home directory and do this (this is based on the current repository setup, which will probably change a little soon):

```
git clone https://github.com/derpycode/ctoaster.cupcake.git
cd ~/ctoaster.cupcake
./setup-ctoaster
```

(Accept the defaults for all the questions it asks – just hit return at each prompt to accept the default.)
Once everything is done, you can run some basic tests with:

```
./tests run basic
```

and you can configure and run jobs as:

```
./new-job -b cgenie.eb_go_gs_ac_bg.p0650e.NONE ...
          ... -u LABS/LAB_0.snowball snowball 10
cd ~/ctoaster.cupcake-jobs/snowball
./go run
```

The job results appear in the `output` directory in the job directory.

There is also a facility to configure and run experiments in the same way as for **cgenie.muffin**:

```
./run-cupcake <base-config> <config-dir> <run-id> <run-length> [<restart>]
```

which for the example above, would be:

```
./run-cupcake cgenie.eb_go_gs_ac_bg.p0650e.NONE LABS LAB_0.snowball 10
```

In this, `run-cupcake.py` first configures a new job using `./new-job`, seamlessly changes directory, and then automatically runs it using `./go run`

[1] *Throughout this document, shell commands are shown in `typewriter` font. Commands that extend over several lines are marked with lines ending `...` with the following line beginning `...`.*

1.2 Installation and setup

To install **cTOASTER**, first choose a location for the installation. (On linux machines, it's probably best just to install **cTOASTER** in your home directory.) Then clone the main `ctoaster.cupcake` repository from GitHub using the command:

```
git clone https://github.com/derpycode/ctoaster.cupcake.git
```

This will produce a new directory called `ctoaster.cupcake` containing the model source code and build scripts.

Before using the model, it's necessary to do a little bit of setup. Go into the new `ctoaster.cupcake` directory and run the `setup-ctoaster` script:

```
cd ~/ctoaster.cupcake
./setup-ctoaster
```

The script will ask where you want to put a number of things – it's usually find to just take the defaults (just hit enter at each of the prompts). In all of what follows below, we'll assume that you chose the defaults. The info the script asks for are:

- The **cTOASTER** root installation directory: unless you know what you're doing, accept the `~/ctoaster.cupcake` default for this.
- The **cTOASTER** data directory (default `~/ctoaster.cupcake-data`) where base and user model configurations are stored, along with forcing files.
- The **cTOASTER** test directory (default `~/ctoaster.cupcake-test`) where **cTOASTER** jobs with known good outputs can be stored for use as tests – it's possible to run sets of tests and compare their results with the known good values with a single command, which is useful for making sure that the model is working.
- The **cTOASTER** jobs directory (default `~/ctoaster.cupcake-jobs`) where new **cTOASTER** jobs are set up – the `new-job` script (see next section) sets jobs up here by default.
- The default model version to use for running jobs. By default, the most recent released version is selected, but you can type another version if necessary. You can also set jobs up to use any model version later on.

After providing this information, the setup script will ask whether you want to download the data and test repositories. It's usually best to say yes.

Once the data and test repositories have been downloaded, **cTOASTER** is ready to use. The setup information is written to a `.ctoasterrc` file in your home directory. If you ever want to set the model up afresh, just remove this file and run the `setup-ctoaster` script again.

To check that the installation has been successful and that the model works on your machine, you can run some basic test jobs – in the `~/ctoaster.cupcake` directory, just type:

```
./tests run basic
```

This runs an ocean-atmosphere simulation and an ocean biogeochemistry simulation using the default model version selected at setup time.

*If you are the first person to use **cTOASTER** on your machine, you will need to set up a platform file to tell **cTOASTER** where to find a **FORTRAN** compiler, what compilation flags to use, and where to find **NetCDF** libraries – see Section~2.7.*

1.3 Creating new jobs

New **cTOASTER** jobs are configured using the `new-job` script in `~/ctoaster.cupcake`. This takes a number of arguments that describe the job to be set up and produces a job directory under `~/ctoaster.cupcake-jobs` containing everything needed to build and run the model with the selected configuration. The `new-job` script should be run as:

```
new-job [options] job-name run-length
```

where `job-name` is the name to be used for the job directory to be created under `~/ctoaster.cupcake-jobs` and `run-length` is the length of the model run in years. The possible options for `new-job` are as follows (in each case given in both short and long forms where these exist). First, there are three options that control the basic configuration of the model. In most cases, a base and a user configuration should be supplied (options `-b` and `-u`). In some special circumstances, a custom "full" configuration may also be used (the `-c` option).

```
-b BASE_CONFIG    --base-config=BASE_CONFIG
```

The base model configuration to use – these are stored in the `~/ctoaster.cupcake-data/base-configs` directory.

```
-u USER_CONFIG    --user-config=USER_CONFIG
```

The user model configuration to apply on top of the base configuration – model user configurations are stored in the `~/ctoaster.cupcake-data/user-configs` directory.

```
-c CONFIG         --config=CONFIG
```

Full configuration name (this is mostly used for conversions of pre-`cupcake` tests) – full configurations are stored in the `~/ctoaster.cupcake-data/full-configs` directory.

In addition to the configuration file options, the following additional options may be supplied to `new-job`:

```
-O      --overwrite
```

Normally, `new-job` will not overwrite any existing job of the requested name. Supplying the `-O` flag causes `new-job` to delete and replace any existing job with the requested name.

```
-r RESTART     --restart=RESTART
```

One **cTOASTER** job can be *restarted* from the end of another. This option allows for a restart job to be specified. This must be a job that has already been run (so that there is output data to use for restarting the model).

```
--old-restart
```

It may sometimes be useful to restart from an old pre-`cupcake` job. This flag indicates that the job name supplied to the `-r` flag is the name of an old **cTOASTER** job whose output can be found in the `~/ctoaster.cupcake_output` directory.

```
--t100
```

This flag indicates that the job should use the alternative "T100" timestepping options for the model (i.e. 100 timesteps per year for the default model resolution instead of 96).

```
-j JOB_DIR      --job-dir=JOB_DIR
```

It can sometimes be useful to put cTOASTER jobs somewhere other than `~/ctoaster.cupcake-jobs`. This flag allows an alternative job directory to be specified.

```
-v MODEL_VERSION    --model-version=MODEL_VERSION
```

Normally, `new-job` will generate a job set up to use the default model version which was selected when the `setup-ctoaster.cupcake` script was run. This flag allows for a different model version to be selected.

Examples

```
./new-job -b cgenie.eb_go_gs_ac_bg.p0650e.NONE ...
          ... -u LABS/LAB_0.snowball snowball 10
```

This configures the first example job in the workshop handout. After running this invocation of `new-job`, a new `~/ctoaster.cupcake-jobs/snowball` job directory will have been created from which the job can be executed.

```
./new-job -b cgenie.eb_go_gs_ac_bg.p0650e.NONE ...
          ... -u LABS/LAB_0.snowball -r snowball snowball2 10
```

This invocation of `new-job` sets up a new `snowball2` job that restarts from the end of the `snowball` job to run for an additional 10 years.

1.4 Running jobs

Once a job has been set up using the `new-job` script, it can be run from the newly created job directory using a "`go`" script. Configuring and running a job is as simple as:

```
cd ~/ctoaster.cupcake
./new-job -b cgenie.eb_go_gs_ac_bg.p0650e.NONE ...
          ... -u LABS/LAB_0.snowball snowball 10
cd ~/ctoaster.cupcake-jobs/snowball
./go run
```

The `go` script has three main options and two advanced options. The basic options are:

`./go clean` Remove model output and model executables and compiled object files for the current job setup.

`./go build` Compile the required version of the model to run this job – this depends on a number of things, including the selected model resolution, but the build system ensures that model executables are not recompiled unnecessarily.

`./go run` Compile the model (if necessary) and run the current job.

Both the `build` and `run` commands can also take a "build type" argument for building debug or profiling versions of the model. For more information about this and about how the build system maintains fresh executables of selected versions of the model, see Section~2. Also see that section for the two "advanced" options to the `go` script, which are used to select alternative "platforms" for a machine – in the normal case, the build system will select the appropriate compilers and flags based on the machine on which the model is being run (assuming that a platform definition has been set up for the machine), but sometimes it may be desirable to select between different compilers on the same machine, for which a `set-platform` option is provided by the `go` script.

1.5 Managing configuration files

Configuration files are all kept in `~/ctoaster.cupcake-data`, base configurations in the `base-configs` directory and user configurations in `user-configs`. All of this configuration data is held in a git repository on **GitHub**, so if you want to add user or base configurations to share with other users, ask someone about how to set yourself up to use **GitHub**.

1.6 Managing tests

It is possible to save job configurations and results as test jobs with "known good" data. This has two main uses – first, for testing a **cTOASTER** installation to make sure that it's working; second, to test that changes to the model don't inadvertently affect simulation results. The second application is of more interest for people changing the **cTOASTER** model code, but it can still be useful to save jobs as tests.

The `tests` script in `~/ctoaster.cupcake` is used to manage and run test jobs. To list the available tests, do

```
./tests list
```

and to run an individual test or a set of tests, do

```
./tests run <test>
```

where `<test>` is either a single test name (e.g. `basic/biogem`), a set of tests (e.g. `basic`) or `ALL`, which runs *all* available tests. The tests are run as normal cTOASTER jobs in a sub-directory of texttt~/ctoaster.cupcake-jobs with a name of the form `test-YYYYMMDD-HHMMSS` based on the current date and time. As well as full test job output, build and run logs, a `test.log` file is produced in this test directory, plus a `summary.txt` file giving a simple pass/fail indication for each test.

An existing job can be added as a test using a command like

```
./tests add <job-name>
```

where <job-name> is the name of an existing job in ~/ctoaster.cupcake-jobs. Note that you need to run the job before you can add it as a test! The test script will ask you which output files you want to use for comparison for each model component – there are sensible defaults in most cases, but you can select individual files too if you prefer.

There are two other features of the test addition command that can be useful. First, it's possible to give the test a different name than the job it's made from – for example

```
./tests add hosing/test-1=hosing-experiment-1
```

adds a test called hosing/test-1 based on the hosing-experiment-1 job. Second, it's possible to say that a new test should be restarted from the output of an existing test. Normally, if a test is created from a job that requires restart files, the restart files are just copied from the job into the new test. Sometimes though, it can be of interest to run the job that generated the restart data, then immediately run a test starting from the output of the first test. This can be done using something like this:

```
./tests add foo/test-1=job-1
./tests add foo/test-2=job-2 -r foo/test-1
```

This indicates that foo/test-1 is a "normal" test, while foo/test-2 is a test that depends on foo/test-1 for its restart data. When you run a test that depends on another for restart data, the test script deals with making sure that the restart test is run before the test that depends on it. So, for example, you can just say

```
./tests run foo/test-2
```

and the test script will figure out that it needs to run foo/test-1 first in order to generate restart data for foo/test-2.

1.7 Managing model versions

For most users, it makes sense to run jobs using the most recent available version of the **cTOASTER** model code. This is the option chosen by default when the model is initially set up. However, it can sometimes be useful to run jobs with earlier model versions (or with a development version of the model – see the next section). The **cTOASTER** configuration and build system provides a simple mechanism to permit this, hiding most of the (rather complex) details of managing multiple model versions from users.

Model versions are indicated by git 'tags'. In order to see a list of available model versions, use the following command in the ~/ctoaster.cupcake directory:

```
git tag -l
```

To configure a job to use a different model version from the default, simply add a -v flag to new-job specifying the model version you want to use. For example, to configure a job to use the cupcake-1.0 version of the model, use something like the following command:

```
./new-job -b cgenie.eb_go_gs_ac_bg.p0650e.NONE ...
          ... -u LABS/LAB_0.snowball snowball 10 ...
          ... -v cupcake-1.0
```

Within a job directory, you can see what model version the job was configured with by looking at the contents of the config/model-version file – in non-development cases, this will just contain the git tag of the model version.

1.8 Legacy (cGENIE) work-flow

The example experiment above can also be configured and run via a single command[2] :

```
./run-cupcake cgenie.eb_go_gs_ac_bg.p0650e.NONE LABS LAB_0.snowball 10
```

This employs run-cupcake.py, which first configures a new job using ./new-job, automatically changes to the corresponding job directory, and then runs it using ./go run.

[2]Usage: run-cupcake <base-config> <config-dir> <run-id> <run-length> [<restart>]

2 For cTOASTER developers

For developers of **cTOASTER**, there are a few extra things to know beyond what's needed to run the model.

2.1 Installation and setup for development

Installation and setup for developers goes almost the same as for users, except that when the `setup-ctoaster` script asks for the default model version to use, you should answer "`DEVELOPMENT`". This causes model executables for all jobs to be built by default from the source code currently in `~/ctoaster.cupcake/src`, rather than from a specified past model version. In this way, you can make changes to the model source code under `~/ctoaster.cupcake/src` and doing a `./go run` in a job directory will trigger a build and execution of the model based on the changed code. (For a simpler way to check for successful model compilation, see below in Section~2.3.)

2.2 Model source organisation

The model source lives in `~/ctoaster.cupcake/src`, with one subdirectory for each model component (`embm`, `biogem`, etc.), the main `cupcake.f90` program, plus a couple of extra sub-directories for utility routines. All of the code is FORTRAN 90 and all source files accordingly have a `.f90` extension.

As well as FORTRAN source files, the per-module and base `src` directories also contain default namelist definitions for each module and "exceptions" files giving mappings from "old" to "new" parameter names that appear in **cTOASTER** configuration files and namelists. All of these things are used by the model configuration system to construct valid namelists from user-specified configuration files.

The other thing you'll see in the source directories are files called `SConscript`. These are the scripts that control the tool used to manage model compilation, described in the next section.

2.3 Build system

The build system uses **SCons**, a 'Make replacement' written in •. The rest of the build system is also written in Python and lives in `~/ctoaster.cupcake/scripts`. In the normal course of things, it shouldn't be necessary for **cTOASTER** model developers to touch this stuff – it should just work. (There are cases where this isn't quite true, mostly to do with major changes in the layout or naming of model input and output files or model parameters, but the scripts have been written as "defensively" as possible, so there shouldn't be *too* many problems.)

Normally the model is compiled and executed from the `go` script in a job directory. This deals with making sure that the correct model version is built from the correct sources (taking account of the model version requested) and again, should just work.

The scripts perform builds in directories under `~/ctoaster.cupcake-jobs/MODELS` – after running a few jobs, you'll see one directory under there for each model version you've used. The directories all ultimately have the form:

```
~/ctoaster.cupcake-jobs/MODELS/<version>/<platform>/<hash>/<build-type>
```

where the different components have the following meanings:

`<version>` The model version for this build. If you run jobs with a non-`DEVELOPMENT` model version, you'll also see a directory called `~/ctoaster.cupcake-jobs/MODELS/REPOS` holding repository copies at fixed versions – when a build is required for a non-`DEVELOPMENT` model version, the source code is accessed from one of these `REPOS` directories rather than from `~/ctoaster.cupcake/src`.

`<platform>` A "platform" is basically a combination of a machine name or type and a compiler/NetCDF directory combination – see Section~2.7. Having this level in the directory structure prevents surprises with home directories NFS-mounted across machines with different compiler or NetCDF path requirements.

`<hash>` One rather undesirable aspect of the current **cTOASTER** setup is that a number of array sizes (mostly coordinate dimension sizes) are compiled into the model as preprocessor constants. This means that jobs using different model grid sizes must use different executables compiled with different preprocessor flags. In order to manage this problem in a coherent way, the preprocessor definitions are collected into a canonical representation and a SHA1 hash is calculated. This provides a unique representation of each model coordinate setup that can be used to segregate model builds.

<build-type> Finally, we distinguish between `ship` (optimised), `debug`, and `profile` builds of the model.

As an example, here's a path to a build directory on my machine (called `seneca`), for an optimised (`ship`) executable for a job with preprocessor definitions `cTOASTERNX=36`, `cTOASTERNY=36`:

```
/home/iross/ctoaster.cupcake-jobs/MODELS/DEVELOPMENT/seneca/...
          ...50b3ce7f3162a0f783e4424c9a294de0061e0cdc/ship
```

The benefit of this arrangement is that it perfectly segregates all the aspects of model configuration, version or build environment that might impact the determination of what source files need to be recompiled at any time. Whenever you type `./go run` or `./go build` in a job directory, only the out of date source files for the exact model version and configuration you need are recompiled, and the object files and executables for that exact configuration are kept seperate from all other model configurations. This avoids any confusing problems with stale files.

Although this is really pretty neat for managing model executables for running jobs, it's not very convenient for day-to-day development, where you often just want to check that the model compiles. In order to make this convenient, the SCons scripts are set up so that it's possible to just run `scons` in the `~/ctoaster.cupcake` directory and have *a* version of the model built right there (actually into a `~/ctoaster.cupcake/build` directory). What this means is that you can set up your editor compilation command (if you use **Emacs**, the thing that gets run when you type `C-c m`) to be "`scons -C ~/ctoaster.cupcake`" and all your usual compiler error message chasing commands will work just right. (The `-C` argument to the `scons` program is the same as that for `make`: it tells SCons to change to a particular directory before running.) If you don't have SCons installed on your machine, you can just run the local version in `~/ctoaster.cupcake/scripts/scons/scons.py`, which should just work.

2.4 Primary workflow

Given the above description, here's the primary workflow I'd recommend for working on **cTOASTER** development (the things to do with git are described in more detail in Section~2.6):

1. Do `git checkout -b <new-branch-name>` to create a git topic branch to work on.
2. Edit files under `~/ctoaster.cupcake/src`.
3. Test model build by running `scons -C ~/ctoaster.cupcake`.
4. Set up test jobs using `new-job`.
5. Run jobs in their job directories using `./go run` (uses development code).
6. Can also run tests, also using development code by default.
7. Build debug and profiling executables using `./go build debug` or `./go build profile` in the job directory. Debug by running `gdb` or equivalent on the `genie-debug.exe` executable directly in the job directory.
8. Commit good changes and push to GitHub – this triggers testing on the Travis continuous integration system. *The Travis stuff isn't set up yet, but it will be!*
9. If the commit passed Travis testing, make a GitHub pull request to have your changes incorporated into the main **cTOASTER** repository.

2.5 Model input parameter handling

The **cTOASTER** executable reads a number of FORTRAN namelists to pick up model parameters. These namelists all have names like `data_EMBM`, `data_BIOGEM`, etc., live in the top level of each job directory and are produced by the `new-job` script by combining values in the base and user configuration files and default namelists for each module.

The `new-job` script is written in a way that should avoid problems with old, no longer used, parameter names appearing in configuration files and with configuration files that do not specify values for newly introduced parameter names. The way that this works is that the default namelists for each module (e.g. `~/ctoaster.cupcake/src/embm/embm-defaults.nml`) contain a default value for every parameter appearing in the relevant namelist in the FORTRAN code. The *new-job* script then ignores any parameters appearing in configuration files whose names do not occur in the default namelist and uses default values for any parameters in the default namelist that do not appear in the configuration files[3]. This means that if you introduce a new parameter or remove an existing parameter from a namelist in the FORTRAN code, you *must* also modify the default namelist file.

2.6 Recommended Git working practices

There are lots of ways of working with git and GitHub. Here are some recommendations based on having used git quite a bit for open source work:

Branches are cheap. Use them! Branches work differently in git than in older version control systems. A branch is just a named state of the repository: switching from one branch to another is very quick and creating new branches costs almost nothing. This enables a different workflow than in a system like Subversion, where branching is a more expensive operation. When using git, you can create branches for *everything*. If you need to make a few small changes to fix a bug, create a branch to do it and merge the changes into the main branch when you're done. If you're working on some more extensive changes, do them on a seperate branch and periodically pull changes to the main branch from the repository and merge them into your working branch so that you don't diverge too much from the main repository (ending up with a huge and complicated merge to do when you've finished). If you work this way, then it's very easy to have a number of tasks going on in parallel without confusion (because the work for each task is on a seperate branch), and you can always switch back to the main branch and make another branch if you need to start work on something else. This "one repository, many branches" style of working is very powerful.

Repository forks and pull requests Instead of just cloning the `ctoaster.cupcake` repository and pushing changes directly to it (which is only possible if you're one of the owners of the repository), create a personal *fork* of the repository on GitHub and do your work in that. You set up the main repository as an *upstream remote* of your personal fork, which allows you easily to pull and changes from the main repository into your fork. And if you do development on branches as suggested above, once you're ready to submit your work for merging into the main repository, you can use GitHub to create a *pull request* for your topic branch. This is an extremely convenient way of communicating changes to the maintainer of the main repository. The maintainer gets an email saying that there's a pull request, the Travis continuous integration tests are run on the pull request branch, and the GitHub user interface provides immediate feedback about whether the pull request is safe to merge. The maintainer can then merge your work into the main repository by pressing a single button.

Issue tracker GitHub incorporates an *issue tracker* for each repository. The is really convenient for keeping track of bugs that need to be fixed or enhancements that people want to have implemented. Each issue has a discussion thread associated with it so developers and maintainers can talk back and forth about what to do. There are also various handy facilities for organising and sorting issues. (Pull requests have the same sort of discussion threads, which can be useful if the work someone submits isn't quite right!)

Releases as tags When the maintainer of a repository wants to create a new official release of a package, they can just create a git tag on the repository. The **cTOASTER** configuration and build system uses these tags to identify the available model versions, and they can also be listed and browsed in the GitHub interface. (In git, a "tag" is just a name for a particular version of the repository so, like branches, they're very lightweight.)

[3] Slight lie: there are some parameters that are set via other mechanisms than the configuration files, notably parameters to do with timestepping and restart files.

2.7 Platform files

The build system has a simple facility for managing compiler paths and options and the location of NetCDF libraries for building **cTOASTER** on different platforms. This mechanism is based on “platform files” in this directory `~/ctoaster.cupcake/platforms`, each of which is named after the hostname of the machine it’s for, or the host and compiler combination (e.g. for my machine `seneca`, something like `seneca-gfortran` or `seneca-ifort`). (There is also a default `LINUX` platform file that uses the GNU Fortran compiler and tries to find NetCDF libraries in some “conventional” places, but that’s really not all that likely to work in most cases...)

Platform files are read when the `go` script for a job needs to build the **cTOASTER** executable. By default, the platform file named after the machine name is used. If you want to use an alternative compiler, use the `set-platform` and `clear-platform` options to the `go` script. For example, on `seneca`, to switch to using the Intel FORTRAN compiler to run a job, do the following:

```
./go set-platform seneca-ifort
./go run
```

and to switch back to using the default for the machine, do:

```
./go clear-platform
./go run
```

Platform files are just Python scripts that set up a few variables. Each platform script must provide definitions for the following names:

`f90` FORTRAN 90 compiler and flag setup. Contains the fields:

- **`compiler`** The name of the compiler executable.
- **`baseflags`** A list of compiler flags to be used for all compilations.
- **`debug`** A list of debug flags (used for build type `debug`).
- **`ship`** A list of optimisation flags (build type `ship`).
- **`profile`** A list of profiling flags (for build type `profile`).
- **`profile_link`** A list of flags to be used for linking builds of type `profile` – many compilers require extra flags to be given at link time for profiling builds.
- **`bounds`** Flags to enable array bounds checking (build type `bounds`).
- **`include`** *Not yet used.*
- **`module_dir`** Flag used by the Fortran 90 compiler to specify the location to write module files.
- **`define`** Flag used to define preprocessor constants.

`netcdf` Options for finding and using NetCDF libraries. Contains the fields:

- **`base`** The base directory of the NetCDF installation. Should contain an `include` subdirectory containing a cTOASTER 90 `netcdf.mod` module file suitable for the compiler being used and a `lib` subdirectory containing the NetCDF libraries.
- **`libs`** A list of NetCDF library names to link the **cTOASTER** executable with. Recent NetCDF installations split the **Fortran** 90 library from the C library, so that one needs to link with both `libnetcdf.a` and `libnetcdff.a` – in this case, this field should have the value `['netcdf', 'netcdff']`; otherwise it should just be `['netcdf']`.

Porting **cTOASTER** to a new platform should require little more than making a new platform file: just copy an existing one, ideally one that uses the same compiler, and edit the locations of the NetCDF libraries. (The platform file should live in `~/ctoaster.cupcake/platforms` and its name should be whatever is returned from the Linux `hostname` command.) It you want to use different compilers on the same platform, just create multiple platform files called `<hostname>-<compiler>` – you’ll probably also want to have a default platform with just the hostname so that you can do builds without setting the platform explicitly. If you want to do builds directly in the `~/ctoaster.cupcake` directory using a non-default compiler, you can either just move the platform files around in `~/ctoaster.cupcake/platforms` so that the default file for your platform uses the compiler you want, or (not really recommended) you can create a file called `~/ctoaster.cupcake/config/platform-name` containing the name of the platform you want to use[4].

[4]This suborns the platform selection method used in builds as run from the `go` script, and the reason it’s not really recommended is

3 For infrastructure developers

I'm not going to write too much here – the best way to understand what's going on in the configuration and build scripts it to read through the `new-job` and `go` scripts and see what they do. However, there are some non-obvious things that deserve a bit of extra explanation.

3.1 Python installation and shell script wrappers

All the configuration and build scripts require Python 3 (and called as `python3`).

The main Python scripts all live in `~/ctoaster.cupcake/tools`.
The `setup-ctoaster`, `new-job`, `go` and `tests` programs are all shell scripts that invoke `python3`.

3.2 Model versions, repositories and development code

The whole story with the `~/ctoaster.cupcake-jobs/MODELS` directory hierarchy for model builds is a little complicated, but it solves a number of related problems:

1. How can users configure run jobs with different model versions in an easy way?
2. How can developers build and run jobs from their development source tree at the same time as being able to run jobs with specific model versions for comparison?
3. How can job configuration information be kept seperate from model source and executable code while maintaining reproducibility of jobs?
4. How can rebuilds be minimised while segregating object and executable files from incompatible model builds?

When a specific model version is required for a job, the `ctoaster.cupcake` source repository is unpacked into a directory under `~/ctoaster.cupcake-jobs/MODELS/REPOS` at exactly that version and the source tree and configuration scripts for that model version are used – this means that even if your main `~/ctoaster.cupcake` directory, for example, is at version `cupcake-3.5`, if you run the `new-job` script telling it to use version `cupcake-1.0` for the new job, *all of the configuration and build steps* for the new job will be done with the `cupcake-1.0` versions of the model and configuration scripts. This allows for perfect reproducibility of jobs between model versions.

The same principle of segregation is applied to platform dependencies and "job hashing". The best way to explain this is with an example. Suppose that you're working on one of the modules of **cTOASTER** making science changes that potentially have platform-dependent effects and that also have effects that depend on model resolution. You've set up a bunch of test jobs with different model resolutions and other characteristics that you use to make sure you don't break things as you make changes. If you change a single FORTRAN file and want to rerun all your tests, how many files need to be recompiled? If you make a new test with a different model resolution, how many FORTRAN files need to be recompiled? In the first case, the answer should be "one, or possibly a few if the file I changed is `USE`d in other files". In the second case, the answer should be "all of them, unless I've already built a model for that resolution recently".

By keeping model builds for different model versions, platforms and model resolutions completely separate, we can make sure that only *exactly* the files that need to be recompiled *are* recompiled in every case. You should never need to "clean" all of the build directories for a model (although the `go` script provides that capability if you really want it) since we maintain exact dependency information at all times.

All of this relies on some careful setup of the SCons `SConstruct` and `SConscript` files, explained a bit more below.

3.3 Job hashing

Because of the way that coordinate sizes for arrays are currently defined in **cTOASTER** (via preprocessor constants), different executables are needed for different grid sizes (and for different numbers of tracers). A simple hashing approach based on the coordinate definitions in each job is used to keep this organized.

---

that it will screw things up if your home directory if NFS-mounted and you try to do builds on a machine different than the one for which you've made the `platform-name` file.

Each job directory has a `job.py` file in its `config` subdirectory. The `job.py` file defines the preprocessor definitions that are needed to build the model version for the job – this file is read as part of the build process to set up the required preprocessor definitions.

When the `go` script needs to work out which directory to use under `~/ctoaster.cupcake-jobs/MODELS` to use for the model for the current job, it reads the `jobs.py` file and turns the coordinate definitions into a canonical form (basically just by stripping whitespace and line endings, and sorting the variable names, producing a string something like "`'cTOASTERNX':36,'cTOASTERNY':36`"). The SHA1 hash of this string is then computed (giving a long string of hex digits and this can then be used as a unique identifier of the job coordinate definition.

3.4 SConstruct organisation

The way that the **cTOASTER** build system using SConstruct is a little bit complicated, mostly in order to manage the segregation of different model builds as described above. The main `~/ctoaster.cupcake/SConstruct` SCons file depends on reading some supporting files from the model build directory (which is either one of the directories under `~/ctoaster.cupcake-jobs/MODELS` or, for the degenerate case of testing model compilation, `~/ctoaster.cupcake`).

Those supporting files are the `job.py` file that defines the model coordinate sizes, described in the previous section, and a file called `version.py` that tells the `SConstruct` script where to find the model source code (`~/ctoaster.cupcake/src` for development builds, or a directory under `~/ctoaster.cupcake-jobs/MODELS/REPOS` for builds using a specified model version) and build scripts and the type of build to perform (e.g. `ship` or `debug`).

If you look in one of the model build directories under `~/ctoaster.cupcake-jobs/MODELS`, you'll find that that's more or less all there is there, apart from the `SConstruct` file, the `build` directory where compilation output goes, a `build.log` file and the `cupcake.exe` executable output. Setting things up this way means that model source code and build results are always kept seperate and there should never be any confusion.

That said, here's a **warning**: the `SConstruct` file uses an SCons feature called "variant directory builds" to pull off the feat of building different model versions in all sorts of different places from the same source tree. The interaction between this feature and SCons's automatic scanning of FORTRAN 90 inter-module dependencies is very very delicate – if you change this in *any* way, I can almost guarantee that you will break it!

3.5 Data file setup for jobs

This is possibly the nastiest and most uncertain part of the model configuration scripts. Because of the way that the namelist parameters are defined for some of the **cTOASTER** model components, it's not possible to determine exactly which model input files (from `~/ctoaster.cupcake/data` or from `~/ctoaster.cupcake-data/forcings`) are required to run a particular job. That's a bit of a problem, since the idea is that each job directory under `~/ctoaster.cupcake-jobs` should be self-contained so that you could tar them up and send them to someone else for them to duplicate the job you were running, or for archiving purposes. This is also important for producing self-contained test cases. To make this possible, each job directory has an `input` subdirectory where all the input files required to run the job live. Getting the required files into that directory requires a little bit of ingenuity.

The `copy_data_files` routine (in `~/ctoaster.cupcake/scripts/config_utils.py`) uses some simple heuristics to figure out what input files might be needed. For each **cTOASTER** component, it extracts a list of candidate filenames from the namelist for that component – basically all string-link parameters that aren't obviously not filenames. It then does three things in order, eliminating candidates that are successfully located before going on to the next step – in each case, if a candidate is matched, the match is copied to the job's `input` directory:

1. Look in the `~/ctoaster.cupcake/data` sub-directory for the relevant model component for an exact match to the candidate.

2. Look in `~/ctoaster.cupcake-data/forcings` for an exact match to the candidate.

3. Look in the `~/ctoaster.cupcake/data` sub-directory for the relevant model component for partial matches to the candidate, i.e. files whose name contain the string we're looking for but aren't an exact match.

The end result of this is that the job's `input` directory ends up containing all the file's that are needed to run the job plus (often) some extraneous files that just happen to have similar names to option values used in the component namelist.

Now, this is pretty ugly and prone to breakage, and is close to the top of my list of candidates for "a better way". The solution I have in mind requires some significant changes to the way that model configurations are set up though, and should probably wait until we have a GUI for configuring **cTOASTER** jobs. In the meantime, this end of things just needs to be tested carefully...

A (Some) teaching labs updated for `cupcake`

A few configuration and data files needed some fixups for these things to work – these changes have all been committed to the `ctoaster.cupcake-data` repository.

A.1 Session #0000

Section 1

```
git clone https://github.com/genie-model/ctoaster.cupcake.git
cd ~/ctoaster.cupcake
./setup-ctoaster.cupcake
./tests run basic
```

Sections 2–6

```
./new-job -b cgenie.eb_go_gs_ac_bg.p0650e.NONE -u LABS/LAB_0.snowball ...
          ... LAB_0.snowball 10
cd ~/ctoaster.cupcake-jobs/LAB_0.snowball
./go run
cd output/biogem
cat biogem_series_ocn_temp.res
```

Section 7

```
cd ~/ctoaster.cupcake_output
wget http://www.seao2.info/ctoaster.cupcake/labs/UoB.2013/...
          ...130328.p0650e.LiCa.OHM10.SPIN0.tar.gz
tar xzf 130328.p0650e.LiCa.OHM10.SPIN0.tar.gz
cd ~/ctoaster.cupcake
./new-job -b cgenie.eb_go_gs_ac_bg.p0650e.NONE -u LABS/LAB_0.snowball ...
          ... LAB_0.snowball-from-restart 100 ...
          ... -r 130328.p0650e.LiCa.OHM10.SPIN0 --old-restart
cd ~/ctoaster.cupcake-jobs/LAB_0.snowball-from-restart
./go run
```

Section 8

```
cd ~/ctoaster.cupcake-data/user-configs
cp LAB_0.snowball LAB_0.snowball-experiment
```

Then edit the `LAB_0.snowball-experiment` configuraton file to change the `ea_radfor_scl_co2` variable (to 10.0, say).

```
cd ~/ctoaster.cupcake
./new-job -b cgenie.eb_go_gs_ac_bg.p0650e.NONE ...
          ... -u LABS/LAB_0.snowball-experiment ...
          ... LAB_0.snowball-experiment-1 100
cd ~/ctoaster.cupcake-jobs/LAB_0.snowball-experiment-1
./go run
```

It's easy to make another job to extend this simulation – just restart from the end of the last job:

```
cd ~/ctoaster.cupcake
./new-job -b cgenie.eb_go_gs_ac_bg.p0650e.NONE ...
          ... -u LABS/LAB_0.snowball-experiment ...
          ... LAB_0.snowball-experiment-1-extend 100 ...
          ... -r LAB_0.snowball-experiment-1
cd ~/ctoaster.cupcake-jobs/LAB_0.snowball-experiment-1-extend
./go run
```

You can do this indefinitely...

A.2 Session #0001

Section 1.3

```
cd ~/ctoaster.cupcake_output
wget http://www.seao2.info/ctoaster.cupcake/labs/UoB.2013/...
          ...EXAMPLE.worjh2.PO4Fe.SPIN.tar.gz
tar xzf EXAMPLE.worjh2.PO4Fe.SPIN.tar.gz
cd ~/ctoaster.cupcake-data/user-configs/LABS
cp LAB_1.colorinjection LAB_1.colorinjection-experiment
```

Edit the `LAB_1.colorinjection-experiment` file as described in the lab script.

```
./new-job -b cgenie.eb_go_gs_ac_bg.worjh2.rb ...
          ... -u LABS/LAB_1.colorinjection-experiment ...
          ... LAB_1.colorinjection 20 ...
          ... -r EXAMPLE.worjh2.PO4Fe.SPIN --old-restart
cd ~/ctoaster.cupcake-jobs/LAB_1.colorinjection
./go run
```

Section 1.5

```
./new-job -b cgenie.eb_go_gs_ac_bg.worjh2.rb ...
          ... -u LABS/LAB_1.hosing LAB_1.hosing 20 ...
          ... -r EXAMPLE.worjh2.PO4Fe.SPIN --old-restart
cd ~/ctoaster.cupcake-jobs/LAB_1.hosing
./go run
```

A.3 Session #0100

Section 1.1

```
./new-job -O -b cgenie.eb_go_gs_ac_bg.worjh2.BASEFe ...
          ... -u LABS/LAB_2.CO2emissions LAB_2.CO2emissions 20 ...
          ...  -r EXAMPLE.worjh2.PO4Fe.SPIN --old-restart
cd ~/ctoaster.cupcake-jobs/LAB_2.CO2emissions
./go run
```

Section 1.2

```
./new-job -O -b cgenie.eb_go_gs_ac_bg.worjh2.BASEFe ...
          ... -u LABS/LAB_2.CONTROL LAB_2.CONTROL 20 ...
          ...  -r EXAMPLE.worjh2.PO4Fe.SPIN --old-restart
cd ~/ctoaster.cupcake-jobs/LAB_2.CONTROL
./go run
```