CASAL2 Contributors Manual

C. Marsh and S. Rasmussen

Contents

1 Introduction **1**

2 Creating a local repository **2**
- 2.1 Git Username and Profile . . . 2
- 2.2 Git Software . . . 2
- 2.3 Cloning a repository . . . 2

3 Maintaining and contributing to a forked repository **4**
- 3.1 Checking the status of the forked repository . . . 4
- 3.2 Updating the forked repository to incorporate changes committed to the master . . . 4

4 Setting up CASAL2 BuildSystem **8**
- 4.1 Overview . . . 8
- 4.2 Building on Windows . . . 9
  - 4.2.1 Prerequisite Software . . . 9
  - 4.2.2 Pre-Build Requirements . . . 10
  - 4.2.3 Building CASAL2 . . . 10
- 4.3 Building on Linux . . . 11
  - 4.3.1 Prerequisite Software . . . 11
  - 4.3.2 Building CASAL2 . . . 11
- 4.4 Troubleshooting . . . 12
  - 4.4.1 Third-party Libraries . . . 12
  - 4.4.2 Main Code Base . . . 12

5 CASAL2 build rules **13**
- 5.1 CASAL2 coding practice and style . . . 13
- 5.2 Unit tests . . . 13
- 5.3 Reporting (Optional) . . . 14
- 5.4 Update manual . . . 15
- 5.5 Builds to pass before merging changes . . . 15

6 Adding Functionality into CASAL2 **17**

7 Merging changes form a forked repository to a master repository **25**

8 Moderating Code for Maintainers **28**

9 Adding Reports **29**

1. Introduction

The Contributors Manual provides an overview for users who wish to access the source code, compile their own version of CASAL2, or who wish to contribute to CASAL2 source code. CASAL2 is open source and the development team encourages users to submit and contribute changes, bug fixes, or enhancements. This manual is intended to provide a guide for users who wish to access the source code for CASAL2, build and compile CASAL2, or contribute to the development of CASAL2.

The CASAL2 source code is hosted on GitHub, and can be found at `https://github.com/NIWAFisheriesModelling/CASAL2`.

A Microsoft Windows bundle which includes a binary, manual, examples and other help guides can be downloaded at `ftp://ftp.niwa.co.nz/Casal2/windows/Casal2.zip` for the Microsoft Windows version, `ftp://ftp.niwa.co.nz/Casal2/windows/casal2_setup.exe` for the Microsoft Windows installer. The Linux bundle which includes a binary, manual, examples and other help guides can be downloaded at `ftp://ftp.niwa.co.nz/Casal2/linux/Casal2.tar.gz`.

If you have any questions or require more information, please contact the CASAL2 development team at casal2@niwa.co.nz.

To maintain the quality and integrity of the code base and ensure that any changes are reliable and accurate, the development team has created this manual to assist contributors in understanding how to access the source code, how to compile and build, and the guidelines for submitting code changes to the CASAL2 Development Team. We want this process to be as pain free as possible to encourage contributions.

This manual covers basics such as setting up a GitHub profile, to using GitHub, and all the way to compiling and modifying source code.

A suggestion to future contributors, before making changes to CASAL2 it may be helpful for you to discuss a future change with the development team before you begin making the change. They can assist with ideas on how to add functionality with ease. The easiest forum to have a conversation on code ideas is through the issues page of GitHub found at `https://github.com/NIWAFisheriesModelling/CASAL2/issues`. Just open up a new issue called, for example, New Migration process and begin by talking about how you were thinking about adding a new idea, and hopefully the development team can assist or make the implementation easier for you.

2. Creating a local repository

This section will cover the following points

1. Registering a GitHub username
2. Download git software
3. Fork the master repository

2.1. Git Username and Profile

The first step is to create a username and profile on GitHub (if you do not already have one). Creating a username and profile on GitHub is free and easy to do.

Go to `https://www.github.com` to register a username and set up a profile if you do not already have one. See the help at GitHub for more information. Once you have set up a GitHub account, you need to download the git software that you will use to "communicate" with repositories (places where the software source code is stored) on GitHub.

2.2. Git Software

You will need to acquire a git client in order to clone a copy of the source code and link this to the repository.

CASAL2 also requires a command line version of git in order to compile. The CASAL2 build environment requires git in order to evaluate the version of the code used at compile time to include into the executable and manual when being built.

One package that allows this on a Microsoft Windows platform is tortoisegit (see `https://tortoisegit.org/download`) to pull, push and commit changes to git repositories. However there are many other clients that could be used to achieve the same functionality.

2.3. Cloning a repository

The publicly available CASAL2 code is in the master repository. Only the CASAL2 Development Team have permission to add, delete, or change code in the master repository. Other contributors can either add, delete, or change code either by forking the master repository' or by maintaining a local version of the repository. Forking a copy on GitHub can be done at `https://github.com/NIWAFisheriesModelling/CASAL2` by selecting the fork button in the top right of the page circled in Figure 2.1

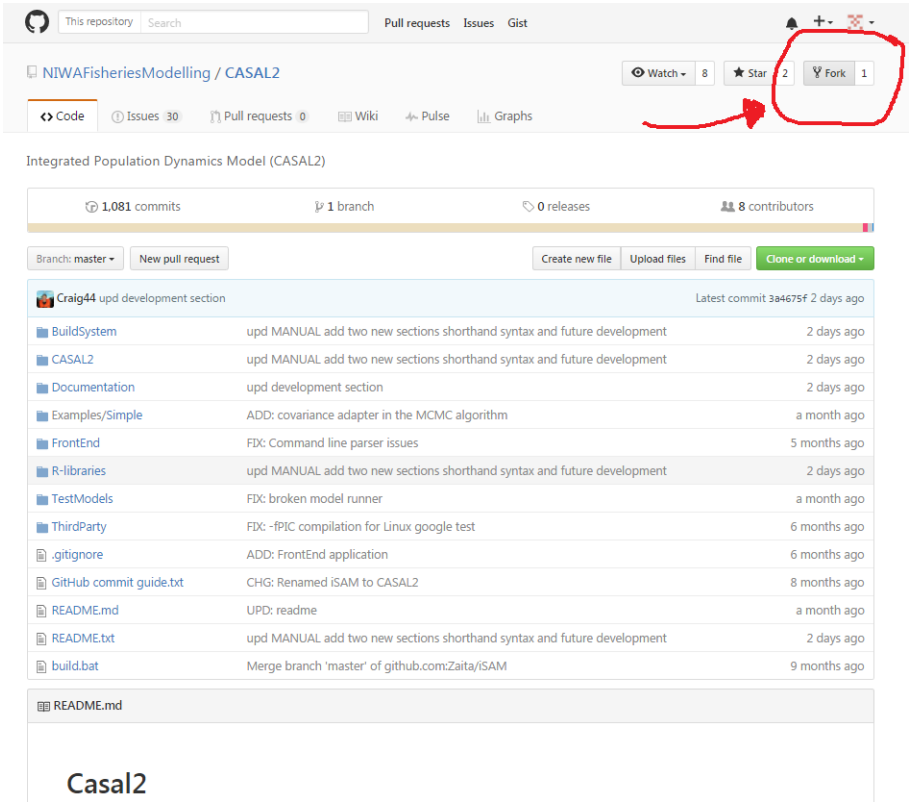

Figure 2.1: Creating a forked repository

This will create a copy of the CASAL2 repository under your profile at the point of the fork. To check that you have successfully forked the repository, go to your git profile and you should see a CASAL2 repository under your repositories, shown in Figure 2.2,

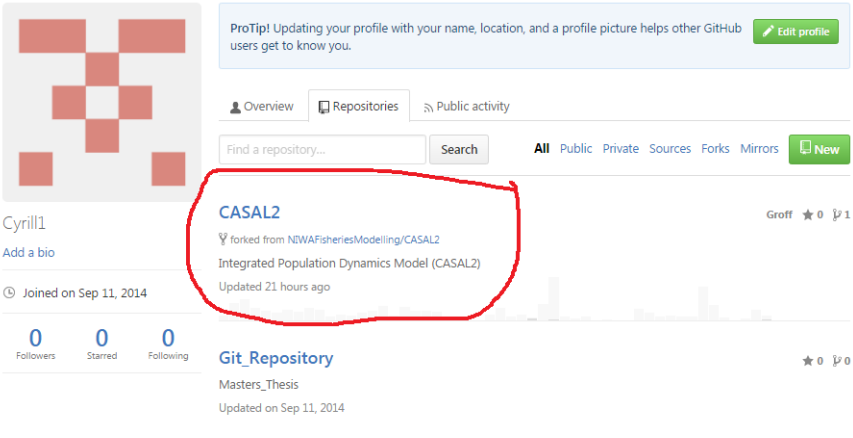

Figure 2.2: Fork success

An important point is that the forked repository will not automatically keep up to date with the master repository. So if the master changes, you will want to keep your forked repository up to date. This can easily be done and is explained in the next section.

3. Maintaining and contributing to a forked repository

This section covers the following details

1. Checking the status of the forked repository compared to the master repository
2. Updating forked repository
3. Making changes to your forked repository

3.1. Checking the status of the forked repository

The status of the forked repository as compared to the master repository can be seen on GitHub see Figure 3.1)

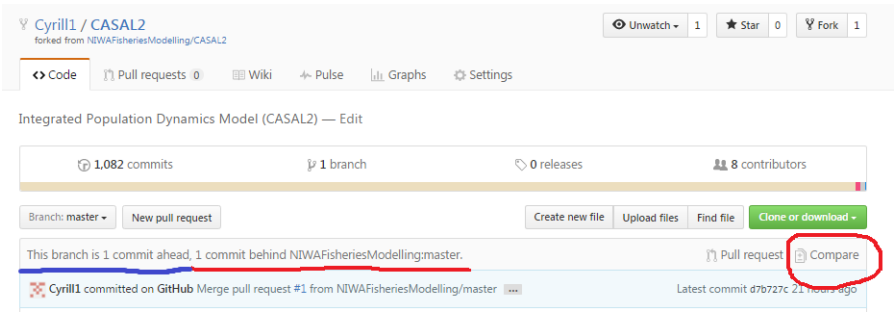

Figure 3.1: Fork status

This line underlined in blue indicates how many commits have occurred on the forked repo since forking the repo or updating the forked repo. The line underlined in red indicates the commits that have occurred to the master since you forked the repo or last updated it. In the above example, we have made one commit to our forked repo (indicated by the blue) and the forked version is one commit behind the master (indicated by the red).

3.2. Updating the forked repository to incorporate changes committed to the master

To update your forked repository to be the same as the master, click the ‘compare’ button (circled in red on the right of Figure 3.1). This will bring up a page that will tell you of the changes that have occurred to the master repository.

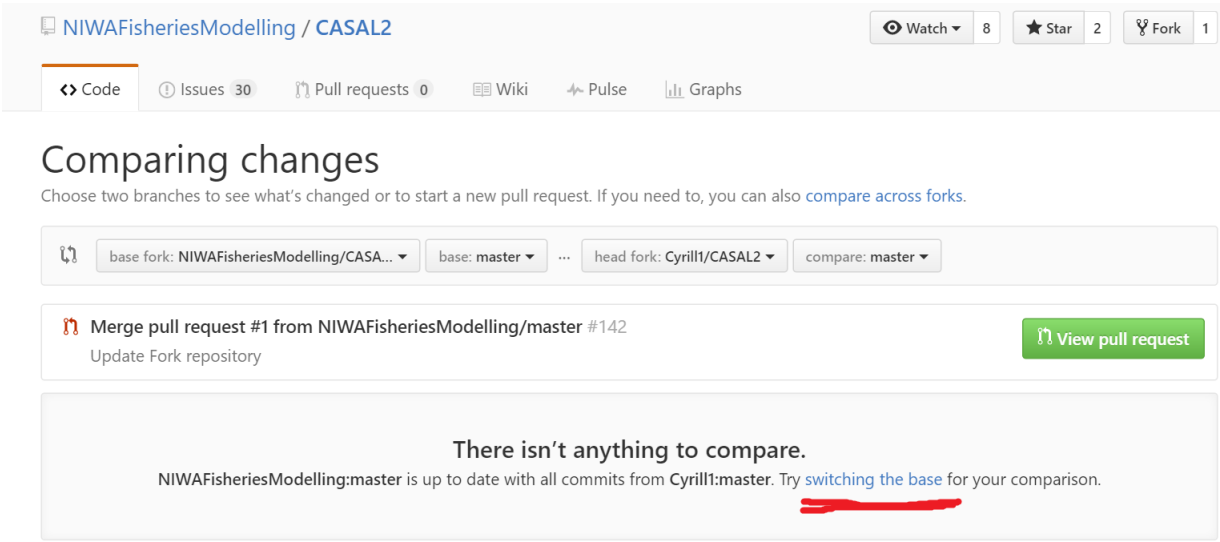

Figure 3.2

If the following figure appears (Figure 3.2) telling you "there isn't anything to compare" click on the `switching the base` underlined in red.

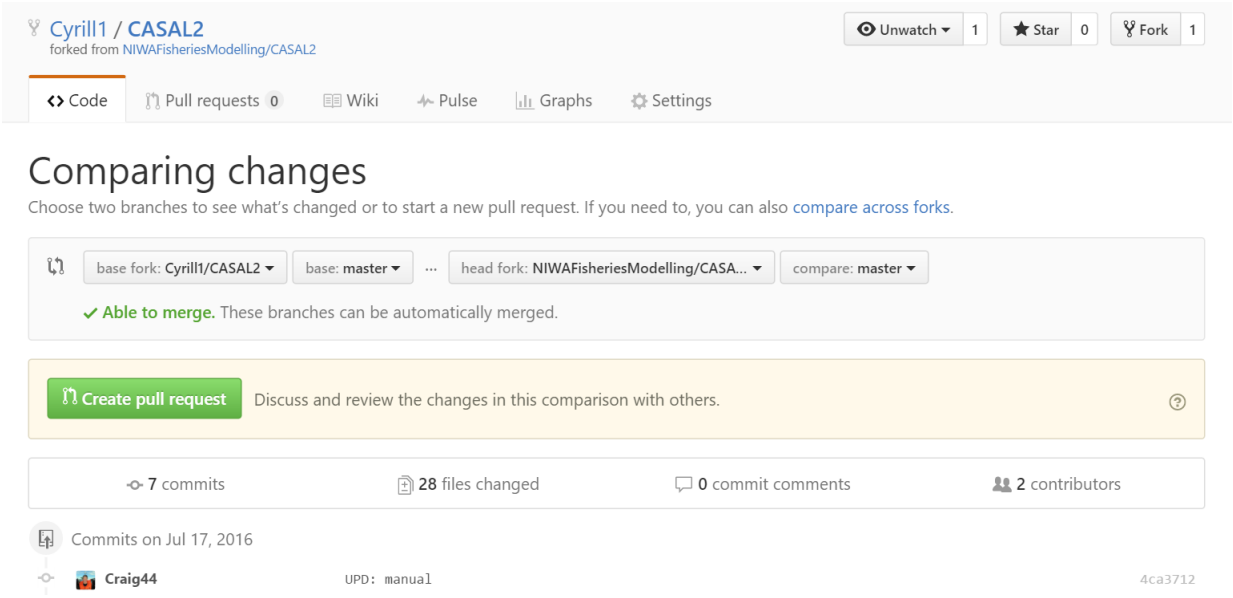

Figure 3.3

You can see that there are 7 commits and 28 files changed on the master that we want to merge into our forked repo. You can see that there is now a "create pull request" and a green message saying "Able to merge". This indicates you will be able to update your forked repo with ease. Click on the "create pull request" button. This will open a git pull request (Figure 3.4) we suggest adding a nice title for the merge like "Updating Craig's Forked Repository"

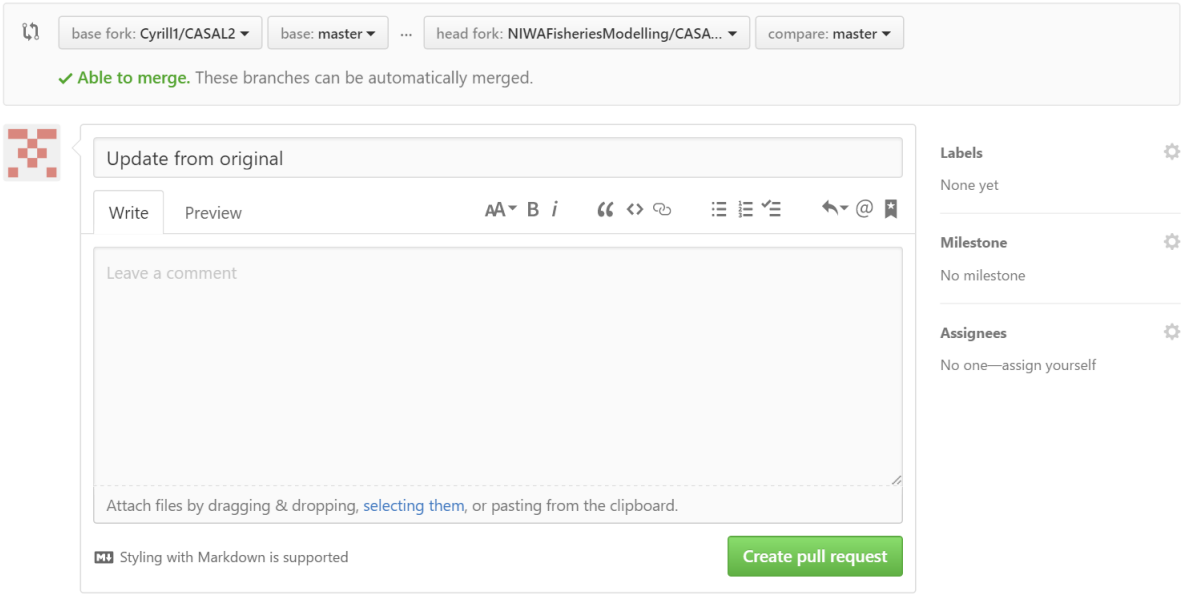

Figure 3.4

Once you write a reasonable title and click create pull request if there are no conflicts, you can merge the changes from the master into your forked repo by clicking "merge pull request" form Figure 3.5.

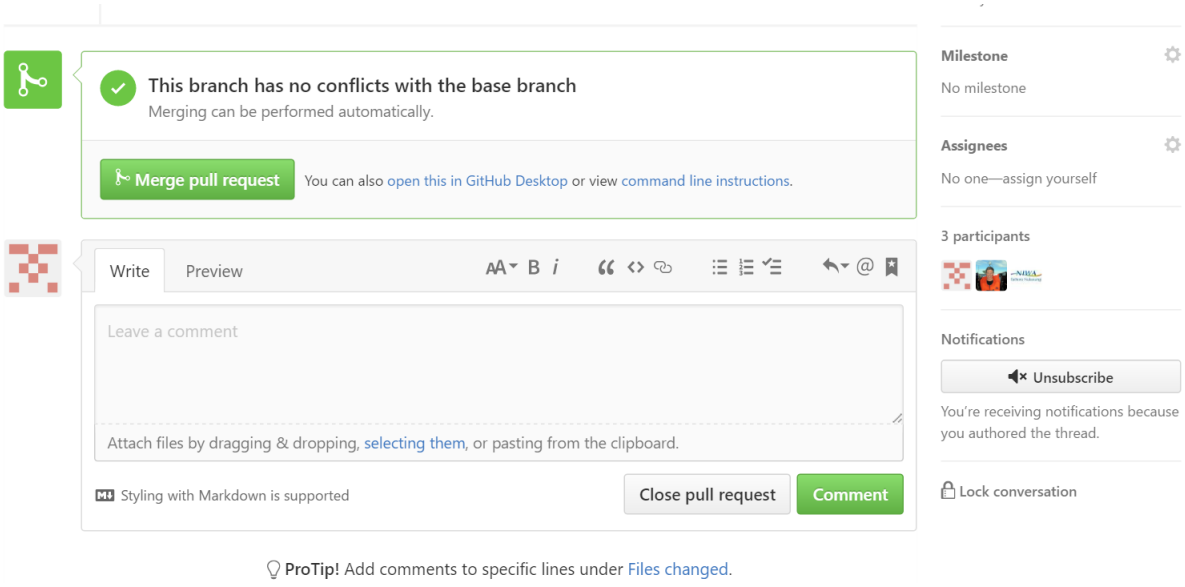

Figure 3.5

Once you have clicked that you will be told that the merge was a success as shown in Figure 3.6.

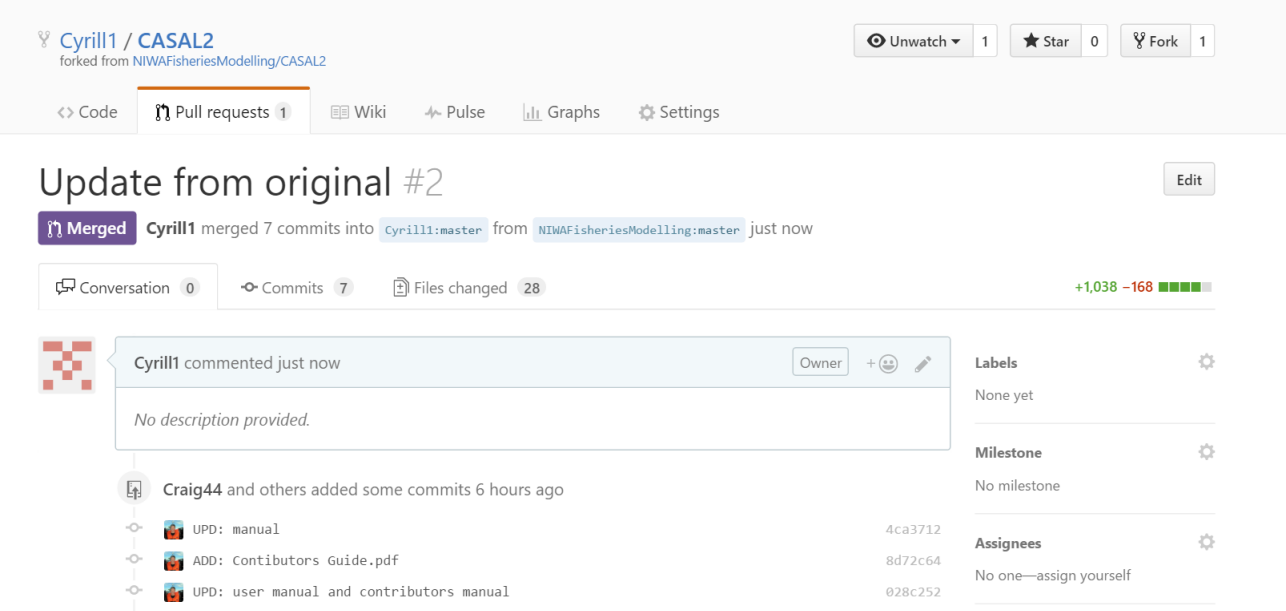

Figure 3.6

Now you have successfully updated your remote forked repository. To incorporate these changes into your local repository on your computer you just need to do a pull and the code changes should be incorporated in your next build.

4. Setting up CASAL2 BuildSystem

This section describes how to set up the environment on your local machine that will allow you to build and compile CASAL2. The build environment can be on either Microsoft Windows or Linux systems. At present the CASAL2 build system supports Microsoft Windows 7+ and Linux (with GCC/G++ 4.9.0+). Apple OSX or other platforms are not currently supported.

4.1. Overview

The build system is made up of a collection of python scripts that do various tasks. These are located in `CASAL2/BuildSystem/buildtools/classes/`. Each python script has it's own set of functionality and undertakes one set of actions.

The top level of the build system can be found at `CASAL2/BuildSystem/`. In this directory you can run `doBuild.bat help` from a command terminal in Microsoft Windows systems or `./doBuild.sh help` from a terminal in Linux systems. - this doesn't make sense. (Craig: What doesn't make sense about it? please give more information if we are to enhance it) The script will take one or two parameters depending on what style of build you'd like to achieve. These commands allow the building of various stand-alone binaries, shared libraries, and the documentation. Note that you will need additional software installed on your system in order to build CASAL2. These requirements are described later.

A summary of all of the doBuild arguments can be found using the command `doBuild help` in the BuildSystem directory.

The current arguments to doBuild are:

- `debug`: Build standalone debug executable
- `release`: Build standalone release executable
- `test`: Build standalone unit tests executable
- `documentation`: Build the user manual
- `thirdparty`: Build all required third party libraries
- `thirdpartylean`: Build minimal third party libraries
- `clean`: Remove any previous debug/release build information
- `cleanall`: Remove all previous build information
- `archive`: Build a zipped archive of the application. The application is built using shared libraries so a single CASAL2executable is created.
- `check`: Do a check of the build system
- `modelrunner`: Run the test suite of models
- `installer`: Build an installer package
- `deb`: Create Linux .deb installer
- `library`: Build shared library for use by front end application

- `frontend`: Build single CASAL2executable with all minimisers and unit tests

Valid Build Parameters: (thirdparty only)

- `<libary name>`: Target third party library to build or rebuild

Valid Build parameters: (debug/release only) e.g. `doBuild.bat release betadiff`

- `betadiff`: Use BetaDiff auto-differentiation (from CASAL)
- `cppad`: Use CppAD auto-differentiation
- `adolc`: Use ADOLC auto-differentiation in compiled executable

Valid Build parameters: (library only)

- `adolc`: Build ADOLC auto-differentiation library
- `betadiff`: Build BetaDiff auto-differentiation library (from CASAL)
- `cppad`: Build CppAD auto-differentiation library
- `test`: Build Unit Tests library
- `release`: Build release library

The outputs from the build system commands will be placed in sub-folders of `CASAL2/BuildSystem/bin/<operatingsystem>/<build_type>`

For example:

`CASAL2/BuildSystem/windows/debug`

`CASAL2/BuildSystem/windows/library_release`

`CASAL2/BuildSystem/windows/thirdparty/`

`CASAL2/BuildSystem/linux/library_release`

4.2. Building on Windows

4.2.1. Prerequisite Software

The building of CASAL2 requires additional build tools and software, including git version control, GCC compiler, LaTeX compiler, and an Windows package builder. CASAL2 requires specific implementations and versions in order to build.

C++ and Fortran Compiler

Source: tdm-gcc (MingW64) from `http://www.tdm-gcc.tdragon.net/`.

CASAL2 is designed to compile under GCC on Microsoft Windows and Linux platforms. While it may be possible to build the package using different compilers, the CASAL2 Development Team does provide any assistance or recommendations. We recommend using 64-bit TDM-GCC version 5.1.0. Ensure you have the "fortran" and "openmp" options installed as a part of the "gcc" dropdown tickboxes otherwise CASAL2 will not compile.. **Note**: A common error that can be made is having

a different GCC compiler in your path when attempting to compile. For example, rtools includes a version of the GCC compiler. We recommend removing these from your path prior to compiling.

GIT Version Control

Source: Command line GIT from `https://www.git-scm.com/downloads`.

CASAL2 automatically adds a version number based on the GIT version of the latest commit to its repository. The command line version of GIT is used to generate a version number for the compiled binaries, R libraries, and the manuals.

MiKTeX Latex Processor

Source: Portable version from `http://www.miktex.org/portable`.

The main user documentation for CASAL2 is a PDF manual generated from LaTeX. The LaTeX syntax sections of the documentation are generated, in part, directly from the code. In order to regenerate the user documentation, you will need the MiKTeX LaTeX compiler.

7-Zip

Source: 7-Zip from `http://www.7-zip.org/download.html`.

The BuildSystem calls 7zip.exe to unzip files in the build system; it is advised to have this in the path.

Inno Setup Installer Builder (optional)

Source: Inno Setup 5 from `http://www.jrsoftware.org/isdl.php`

If you wish to build a Microsoft Windows compatible Installer for CASAL2 then you will need the Inno Setup 5 application installed on the machine. The installation path must be `C:\ProgramFiles(x86)\InnoSetup5\` in order for the build scripts to fins and use it.

4.2.2. Pre-Build Requirements

Prior to building CASAL2 you will need to ensure you have both G++ and GIT in your path. You can check both of these by typing:

```
g++ -version
```

```
git -version
```

This also allows you to check that there are no alternative versions of a GCC compiler that may confuse the CASAL2 build.

ItâĂŹs worth checking to ensure GFortran has been installed with the G++ compiler by typing:

```
gfortran -version
```

If you wish to build the documentation bibtex will also need to be in the path:

```
bibtex -version
```

4.2.3. Building CASAL2

The build process is relatively straightforward. You can run `doBuild check` to see if your build environment is ready.

1. Get a copy (clone) of the forked code on your local machine, mentioned in Section 2:

2. Navigate to the BuildSystem folder in `CASAL2/BuildSystem`
3. You need to build the third party libraries with:
    - `doBuild thirdparty`
4. You need to build the binary you want to use:
    - `doBuild release`
5. You can build the documentation if you want:
    - `doBuild documentation`

4.3. Building on Linux

This guide has been written against a fresh install of Ubuntu 15.10. With Ubuntu we use apt-get to install new packages. YouâĂŹll need to be familiar with the package manager for your distribution to correctly install the required prerequisite software.

4.3.1. Prerequisite Software

Compiler G++

Ubuntu 15.10 comes with G++ 15.10, gfortran is not installed though so we can install it with: `sudo apt-get install gfortran`.

GIT Version Control

Git isn't installed by default but we can install it with `sudo apt-get install git`

CASAL2 automatically adds a version number based on the GIT version of the latest commit to its repository. The command line version of GIT is used to generate a version number for the compiled binaries, R libraries, and the manuals.

CMake

CMake is required to build multiple third-party libraries and the main code base. You can do this with `sudo apt-get install cmake`

Python2 Modules

There are a couple of Python2 modules that are required to build CASAL2. These can be installed with `sudo apt-get install python-dateutil`

You may also need to install **datetime**, re and **distutils**. **Texlive** Latex Processor. No supported latex processors are installed with Ubuntu by default. You can install a suitable latex process with:

```
sudo apt-get install texlive-binaries  sudo apt-get install texlive-latex-base
sudo apt-get install texlive-latex-recommended           sudo apt-get install
texlive-latex-extra
```

Alternatively you can install the complete package: `sudo apt-get install texlive-full`

4.3.2. Building CASAL2

The build process is relatively straightforward. You can run `./doBuild.sh check` to see if your build environment is ready.

1. Get a copy (clone) of the forked code on your local machine, mentioned in Section 2:
2. Navigate to the BuildSystem folder in `CASAL2/BuildSystem`
3. You need to build the third party libraries with:
   - `./doBuild.sh thirdparty`
4. You need to build the binary you want to use:
   - `./doBuild.sh release`
5. You can build the documentation if you want:
   - `./doBuild.sh documentation`

4.4. Troubleshooting

4.4.1. Third-party Libraries

It's possible that there will be build errors or issues building the third-party libraries. If you encounter an error, then itâĂŹs worth checking the log files. Each third-party build system stores a log of everything itâĂŹs doing. The files will be named

- casal2_unzip.log
- casal2_configure.log
- casal2_make.log
- casal2_build.log
- ...etc,.

Some of the third-party libraries require very specialised environments for compiling under GCC on Windows. These libraries are packaged with MSYS (MinGW Linux style shell system). The log files for these will be found in `ThirdParty/<libraryname>/msys/1.0/<libraryname>/`

e.g: `ThirdParty/adolc/msys/1.0/adolc/ADOL-C-2.5.2/casal2_make.log`
e.g: `ThirdParty/boost/boost_1_58_0/casal2_build.log`

4.4.2. Main Code Base

If the unmodified code base does not compile, the most likely cause is an issue with the third-party libraries not being built. Ensure they have been built correctly. As they are outside the control of the Development Team, problems can arise that may require the developers of the third party libraries to resolve first. Contact the CASAL2 development team at `casal2@niwa.co.nz` for help.

5. CASAL2 build rules

This section describes the standards and requirements for code to be included within the CASAL2 code base.

5.1. CASAL2 coding practice and style

CASAL2 is written in C++ and follows the Google C++ style guide (see `https://google.github.io/styleguide/cppguide.html`). The guide is long and comprehensive, so we don't necessarily recommend that you read or understand all of its content. However, the Development Team would like you to follow the Google style of code layout and structure. Google provides a handy script to parse source code and highlight errors that do not match their coding style.

This means using good indentations for functions and loops, sensible (human readable) variable names but noting the use of the characters '_' on the end of class variables defined in the .h files.

Annotate your code. For readability we encourage you to put lots of comments in your code. This helps others read what you intended.

On top of annotating your code we encourage developers to add descriptive logging statements (print messages) in the code. You will see this in the source code already. The purpose of this is to allow a descriptive summary of the actions being done in the model for debugging purposes and curious users. By using these, it becomes easier to identify issues, errors and creates a transparent model for users.

You can also output the text and equations in these logs that would normally be too detailed for general users to be interested, as this may allow users to verify the exact equations or processes that have been implemented. These are really there for curious users and developers.

There are different levels of logging in CASAL2 listed below.

- LOG_MEDIUM() usually reserved for iterative functionality (e.g. estimates during estimation phase)
- LOG_FINE() the level of reporting between an actual report and a fine scale detail that end users are not interested in (Developers)
- LOG_FINEST() Minor details
- LOG_TRACE() put at the beginning of every function

To run CASAL2 in log mode piping out any LOG_FINEST and coarser logs (LOG_MEDIUM and LOG_FINE) you can use the following command,

```
casal2 -r -loglevel finest > MyReports.csl2 2> log.out
```

This will output all the logged information to `log.out`.

5.2. Unit tests

One of the key focusses in the CASAL2 development is an emphasis on software integrity — this is to help ensure that the results from implemented models are consistent and error free. As part of this, we use unit tests to check the individual components of the code base, as well as tests that run entire models in order to validate that the outputs are what we expect them to be.

CASAL2 uses:

- Google testing framework
- Google mocking framework

When adding unit tests, they need to be developed and tested outside of CASAL2 first, for example in **R** or another program like CASAL2 e.g. CASAL or Stock Synthesis. This gives confidence that the test does not contain a calculation or other errors. There are three different testing concepts in CASAL2 that I will quickly explain and point to examples in the code base. The three concepts are Mocking specific classes, implementing internal models (these two concepts are implemented in source files with extension `.Test.cpp`) that have variable test cases for a specific class and overall models found in the TestModel folder.

Mocking specific classes is used to validate specific functionality and is encouraged because it is the easiest to isolate errors introduced with new code changes. Good example in the code base that utilise this are in `AgeLengths` for the `VonBertalanffy.Test.cpp` file. You can see in this file we mock the class and test the mean length calculations. Another good example of this is in the `Partition` class file `Partition.Test.cpp` which shows how CASAL2 validate user inputs and model expectations.

Implementing internal models is the most common test resource in the code, and much of the functionality for implementing these are in the source folder `TestResources`. This method implements simple models and run test cases with differing class implementations. Nice examples of this are in the `Projects` class, for example `LogNormal.Test.cpp`. These run an Internal empty model and test output of classes from a simple run. When using this method try and only add test cases that vary the class of interest, remember you trying to tell future developers if code they have added breaks other components and they will thank you if it is easy to find why this happens. Another example is the process `TagByLength.Test.cpp`, which defines a new model and tests different functionality of the tagging process.

Modelunner tests are run using the `DoBuild` script in the BuildSystem folder. Models are located in the `TestModel` folder. These models are used to test autodiff library model runs, and other functionality such `-i` and are more holistic tests. If these fail it can be difficult to isolate the error that is why we encourage you implement the other two unit tests as much as possible. The setup for the modelrunner are defined in the python script found at `BuildSystem/BuildTools/classes/ModelRunner.py` if you wanted to add simulation TestModels and projections.

An example of how to add a unit test for a process is shown in Section 6

5.3. Reporting (Optional)

Currently CASAL2 has reports that are **R** compatible, i.e., all output reports produced by CASAL2 can be read into **R** using the standard **CASAL2 R** package. If you create a new report or modify an old one, you most follow the standard so that the report is **R** compatible.

All reports must start with, `*label (type)` and end with, `*end`

Depending on what type of information you wish to report, will depend on the syntax you need to use. For example

{d} (Dataframe)

Report a dataframe

```
*estimates (estimate_value)
values {d}
process[Recruitment_BOP].r0 process[Recruitment_ENLD].r0
2e+006 8e+006
*end
```

{m} (Matrix)

Report a matrix

```
*covar (covariance_matrix)
Covariance_Matrix {m}
2.29729e+010 -742.276 -70160.5
-110126 -424507 -81300
-36283.4 955920 -52736.2
*end
```

{L} (List)

Report a List

```
*weight_one (partition_mean_weight)
year: 1900
ENLD.EN.notag {L}
mean_weights {L}
0.0476604 0.111575 0.199705
end {L}
age_lengths {L}
12.0314 16.2808 20.0135
end {L}
end {L}
*end
```

5.4. Update manual

The syntax sections of the user manual are automatically generated from the source code. This means contributors will need to add or modify the remaining sections of the the user manual to document their changes. This is a requirement of contributed or suggested code changes, and is important for end users to be able to use the new or modified functionality.

5.5. Builds to pass before merging changes

Once you have made changes to the code, you must run the following builds before your changes can be considered for inclusion in the the main code base.

build the unittest version see Section 4 for how to build unittest depending on your system.

Run the standard and new unit tests to check that they all pass, to do this first compile the test executable using the script `DoBuild test`. Then move to the directory with the location of the

executable (`BuildSystem/bin/OS/test`) and run it (open a command terminal and run `casal2`) to check all the unit-tests pass.

And test that the debug and release of CASAL2 compile and run. `DoBuild debug`

Then run the second phase of unit tests (requires the debug version is built). This runs the tests that comprise of complete model runs `DoBuild modelrunner`

Build the archive `DoBuild archive` which builds all autodiff libraries. There are small nuances between Double classes, especially when reporting the class that mean seemingly simple changes can sometimes cause a break in the full build.

6. Adding Functionality into CASAL2

The following shows a sequence of figures and text of an example, where we demonstrate how to modify the code base by adding a new child process called SurvivalConstantRate. Although we will be showing adding new process functionality for a process we will explain it in a generalised way so that this can be translated into adding a new selectivity, observation, likelihood, Projection, MCMC, minimiser, time varying class etc.

To add a new process you start by going into the `CASAL2\CASAL2\source\processes\`.

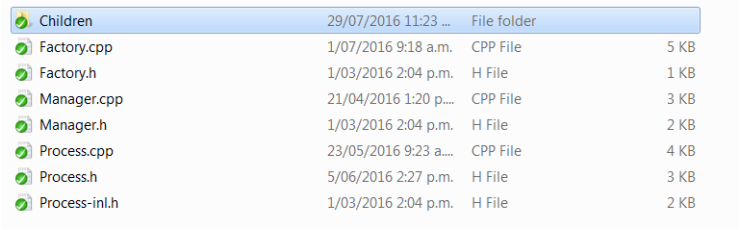

Figure 6.1

This folder shown in Figure 6.1 contains a general layout of all main classes in CASAL2. It contains a folder named `Children`, C++ source code `Factory.cpp`, `Factory.h`, `Manager.cpp`, `Manager.h`, `Process.cpp`, `Process.h`. The folder `Children` contains all the current processes implemented in CASAL2. The `Factory` source code creates all the processes during runtime. The `Manager` source code manages the class; it can build pointers to specific processes to share information across classes. The `Process` source code contains base functionality that is inherited by all child classes (inheritance is a major concept in C++ and it is advised have some knowledge of the concept). Before creating a new child look at the parent (in this example `Process.cpp`, `Process.h`), to see the functionality you don't have to add in your child because it is already done at the parent level. The following figure shows the constructor in `Process.cpp`.

```cpp
#include "Model/Managers.h"
#include "Model/Model.h"
#include "Reports/Manager.h"
#include "Reports/Children/Process.h"

// namespaces
namespace niwa {

/**
 * Default constructor
 */
Process::Process(Model* model) : model_(model) {
  parameters_.Bind<string>(PARAM_LABEL, &label_, "Label", "");
  parameters_.Bind<string>(PARAM_TYPE, &type_, "Type", "", "");
  parameters_.Bind<bool>(PARAM_PRINT_REPORT, &create_report_, "Generate parameter report", "", false);

}

/**
 * Call the validation method for the child object of this process and
 * set some generic variables.
 */
void Process::Validate() {
  parameters_.Populate();

  if (block_type_ != PARAM_PROCESS && block_type_ != PARAM_PROCESSES) {
    if (type_ != "")
      type_ = block_type_ + "_" + type_;
    else
      type_ = block_type_;

    block_type_ = PARAM_PROCESS;
  }

  if (process_type_ == ProcessType::kUnknown)
    LOG_CODE_ERROR() << "process_type_ == ProcessType::kUnknown for label: " << label();

  DoValidate();
}
```

Figure 6.2

From Figure 6.2 the process parent class does not do much - it assigns a label and type for each child that will be created and a report subcommand. The point of this is to look at the parent class to reduce duplicating functionality in the child class.

We will be returning to the factory source code later, but for now let's get adding a new process. Enter the `Children` folder and create C++ source code labelled `SurvivalConstantRate.cpp` and `SurvivalConstantRate.h`. These files were should exist in the source code, but fell free to delete them and go through the process of adding the process to see how easy this process is. I usually copy an existing process and rename it, although this is not good coding practice. It's one of those do-as-I-say-not-as-I-do types of things.

RecruitmentBevertonHolt.Test.cpp	25/07/2016 7:50 a....	CPP File	9 KB
RecruitmentConstant.cpp	1/03/2016 2:04 p.m.	CPP File	5 KB
RecruitmentConstant.h	1/03/2016 2:04 p.m.	H File	2 KB
RecruitmentConstant.Test.cpp	1/03/2016 2:04 p.m.	CPP File	3 KB
SurvivalConstantRate.cpp	29/07/2016 11:23 ...	CPP File	6 KB
SurvivalConstantRate.h	29/07/2016 11:23 ...	H File	2 KB
TagByAge.cpp	1/03/2016 2:04 p.m.	CPP File	18 KB
TagByAge.h	1/03/2016 2:04 p.m.	H File	3 KB
TagByAge.Test.cpp	24/07/2016 10:00 ...	CPP File	14 KB
TagByLength.cpp	15/07/2016 1:26 p....	CPP File	16 KB

Figure 6.3

Once you have done that for each child class you may have to write code for the following functions: `DoValidate()`, `DoBuild()`, `PreExecute()`, `DoExecute()`, `PostExecute()`, `DoReset()`. To understand what these all do Figure 6.4 shows the state transition of CASAL2.

Now we describe the purpose of each transition and this will give you an idea of what you should be incorporating into each function.

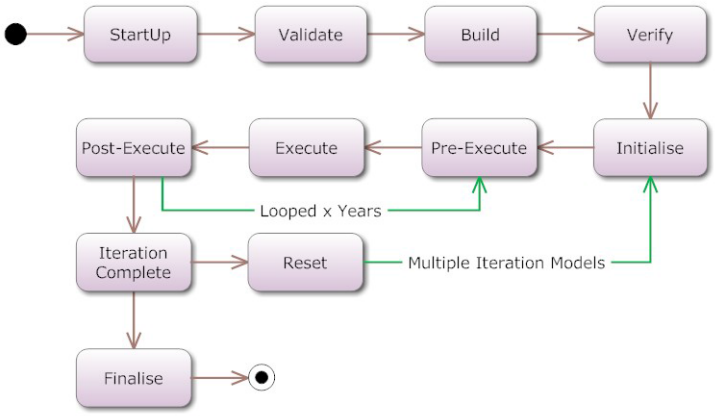

Figure 6.4

StartUp

The model is in the blank start and the configuration system is loading the configuration files and parsing any extra inputs.
Tasks completed:

- Parse command line
- Parse configuration file
- Load plugins
- Load estimate values from input files

Validate

All user configurations have been loaded at this point. Now the model will go through every object that has been created and check that the parameters given to them.

This step will ensure every object in the model has sufficient parameters to be executed without causing a system fault or error in the model.

This state will not check the values to ensure they are logical in relation to an actual model. They will only test that they exist and meet minimum requirements to execute a model.

At the end of the validate stage each object should be internally consistent. No lookups or external references are allowed to be formed during this stage.

Build

The build phase is where the system will build relationships between objects that rely on each other. Because validation has been completed, each object in it's self-contained configuration is ok.

This phase generally assigns values to pointers for objects so they don't need to do lookups of objects during execution phases.

Verify

At this point pre-defined configurations are checked against the model's current configuration to verify if the model makes sense logically. These are business rules being applied to the model to help ensure the output is not garbage.

Note: This has not been implemented.

PreExecute

Pre-Execution happens at the beginning of a time step. This allows objects to calculate values based on the partition state before any of the other processes in the time step are executed.

Execute

This is the general work method of the model and where all of the processes will be run against the partition.

PostExecute

This is executed at the end of a time step after all of the processes and associated objects have been executed. This is typically used for things like reports and derived quantities.

IterationComplete

This is executed at the end of every model run. This is only useful when the model is in a multiple-iteration mode (e.g MCMC or Estimation). After every model iteration this state is triggered.

Reset

If the model has to run multiple iterations then the reset state is used to reset everything back in to a state where the model can be re-executed without any legacy data remaining.

This state allows us to run multiple iterations of the model without having to re-process the configuration information or de-allocate/re-allocate large amounts of memory.

Finalise

Finalise will happen after all iterations of the model have been completed.

Coming back to our example, if we look in `SurvivalConstantRate.h` we will see the following setup, as shown in Figure 6.5.

```
// namespaces
namespace niwa {
class Selectivity; // Forward declare Selectivity class

namespace processes {
namespace accessor = niwa::partition::accessors;
using utilities::OrderedMap;
/**
 * Class Definition
 */
class SurvivalConstantRate : public niwa::Process {
public:
  // Methods
  SurvivalConstantRate(Model* model);
  virtual                     ~SurvivalConstantRate() = default;
  void                        DoValidate() override final;
  void                        DoBuild() override final;
  void                        DoReset() override final;
  void                        DoExecute() override final;

private:
  // Members
  vector<string>              category_labels_;
  vector<Double>              s_input_;
  OrderedMap<string, Double>  s_;
  vector<Double>              ratios_;
  map<unsigned, Double>       time_step_ratios_;
  vector<vector<Double>>      survival_rates_;
  vector<string>              selectivity_names_;
  accessor::Categories        partition_;
  vector<Selectivity*>        selectivities_;
};
```

Figure 6.5

We can see that we are implementing `DoValidate()`, `DoBuild()`, `DoExecute()`, `DoReset()`, which are public functions and we define our variables as private. This idea of structuring classes by public, private and protected is known as encapsulation, and is another important C++ concept. Our variables will be stored in memory for the entire model run so you must make decisions on whether a variable needs to persist for the entire model run or can be temporarily created and destroyed at execution time (Note: the _ at the end this indicates whether the variable is defined in the `.h` file or not).

Moving into `SurvivalConstantRate.cpp`, if we look at the constructor, we can see that this describes input parameters that are supplied by the user, and which variables are estimable. This is applied by the `RegisterAsEstimable()` function.

```
SurvivalConstantRate::SurvivalConstantRate(Model* model)
  : Process(model),
    partition_(model) {
  LOG_TRACE();
  process_type_ = ProcessType::kMortality;
  partition_structure_ = PartitionStructure::kAge;

  parameters_.Bind<string>(PARAM_CATEGORIES, &category_labels_, "List of categories", "");
  parameters_.Bind<Double>(PARAM_S, &s_input_, "Survival rates", "");
  parameters_.Bind<Double>(PARAM_TIME_STEP_RATIO, &ratios_, "Time step ratios for S", "", true);
  parameters_.Bind<string>(PARAM_SELECTIVITIES, &selectivity_names_, "Selectivity label", "");

  RegisterAsEstimable(PARAM_S, &s_);
}
```

Figure 6.6

When you see keywords that begin with `PARAM`, for example `PARAM_CATEGORIES`, these will be defined in `CASAL2\CASAL2\source\English_UK.h`.

Moving on to the `DoValidate()` function in Figure 6.7. The purpose of this function is to validate

the user's inputs and expand inputs to allow short hand syntax. You can see the annotation of what is happening in the function. This is considered one of the coding commandments **Thou shall annotate thy code**.

```
/**
 * Validate our Survival Constant Rate process
 *
 * - Validate the required parameters
 * - Assign the label from the parameters
 * - Assign and validate remaining parameters
 * - Duplicate 's' and 'selectivities' if only 1 vale specified
 * - Check s is between 0.0 and 1.0
 * - Check the categories are real
 */
void SurvivalConstantRate::DoValidate() {
  category_labels_ = model_->categories()->ExpandLabels(category_labels_, parameters_.Get(PARAM_CATEGORIES));

  // If one S supplied expand for each category
  if (s_input_.size() == 1)
    s_input_.assign(category_labels_.size(), s_input_[0]);
  // Do the same for selectivity labels
  if (selectivity_names_.size() == 1)
    selectivity_names_.assign(category_labels_.size(), selectivity_names_[0]);
  //Check we have equal category labels as survival rates
  if (s_input_.size() != category_labels_.size()) {
    LOG_ERROR_P(PARAM_S)
        << ": Number of Ms provided is not the same as the number of categories provided. Expected: "
        << category_labels_.size()<< " but got " << s_input_.size();
  }
  //Check we have equal category labels to selectivity labels
  if (selectivity_names_.size() != category_labels_.size()) {
    LOG_ERROR_P(PARAM_SELECTIVITIES)
        << ": Number of selectivities provided is not the same as the number of categories provided. Expected: "
        << category_labels_.size()<< " but got " << selectivity_names_.size();
  }

  // Validate our S's are between 1.0 and 0.0
  for (Double s : s_input_) {
    if (s < 0.0 || s > 1.0)
      LOG_ERROR_P(PARAM_S) << ": s value " << AS_DOUBLE(s) << " must be between 0.0 and 1.0 (inclusive)";
  }
  // Assign survival rates to a map s_
  for (unsigned i = 0; i < s_input_.size(); ++i)
    s_[category_labels_[i]] = s_input_[i];

  // Check categories are real
  for (const string& label : category_labels_) {
    if (!model_->categories()->IsValid(label))
      LOG_ERROR_P(PARAM_CATEGORIES) << ": category " << label << " does not exist. Have you defined it?";
  }
}
```

Figure 6.7

Another nice inclusion to the CASAL2 source code is it's error checking and messages to the users. The following list is of all the allowable types of errors that CASAL2 can perform.

- `LOG_WARNING() « "warning message";` if triggered print the warning message at the end of model run to, warn the user of a change or implication in the model.
- `LOG_ERROR() « "error message";` After validate and build stop execution and print error message.
- `LOG_ERROR_P(PARAM_KEYWORD) « "error message";` After validate and build stop execution and print error message with location of the parameter `PARAM_KEYWORD` in the configuration files.
- `LOG_FATAL()` if triggered halt execution and print error message.

- `LOG_FATAL_P(PARAM_KEYWORD) « "error message";` If triggered stop execution and print error message with location of the `PARAM_KEYWORD` in the configuration files.
- `LOG_CODE_ERROR() « "error message";` if triggered quit CASAL2 with a message like contact the development team this is a bug in the software.

We encourage you to put these all throughout the validate and build functions, to help users correctly specify models and catch silly input sequences.

Moving on to the `DoBuild()` function. The purpose of the build function is to create pointers and relationships with other classes that will be required during execution. For example, in this process we will require access to the partition, time step and selectivity classes.

```
/**
 * Build any runtime relationships
 * - Build the partition accessor
 * - Build our list of selectivities
 * - Build our ratios for the number of time steps
 */
void SurvivalConstantRate::DoBuild() {
  partition_.Init(category_labels_);

  for (string label : selectivity_names_) {
    Selectivity* selectivity = model_->managers().selectivity()->GetSelectivity(label);
    if (!selectivity)
      LOG_ERROR_P(PARAM_SELECTIVITIES) << ": selectivity " << label << " does not exist. Have you defined it?";

    selectivities_.push_back(selectivity);
  }

  /**
   * Organise our time step ratios. Each time step can
   * apply a different ratio of S so here we want to verify
   * we have enough and re-scale them to 1.0
   */
  vector<TimeStep*> time_steps = model_->managers().time_step()->ordered_time_steps();
  LOG_FINEST() << "time_steps.size(): " << time_steps.size();
  vector<unsigned> active_time_steps;
  for (unsigned i = 0; i < time_steps.size(); ++i) {
    if (time_steps[i]->HasProcess(label_))
      active_time_steps.push_back(i);
  }

  if (ratios_.size() == 0) {
    for (unsigned i : active_time_steps)
      time_step_ratios_[i] = 1.0;
  } else {
    if (ratios_.size() != active_time_steps.size())
      LOG_ERROR_P(PARAM_TIME_STEP_RATIO) << " length (" << ratios_.size()
          << ") does not match the number of time steps this process has been assigned to (" << active_time_steps.size() << ")";

    for (Double value : ratios_) {
      if (value <= 0.0 || value > 1.0)
        LOG_ERROR_P(PARAM_TIME_STEP_RATIO) << " value (" << value << ") must be between 0.0 (exclusive) and 1.0 (inclusive)";
    }

    for (unsigned i = 0; i < ratios_.size(); ++i)
      time_step_ratios_[active_time_steps[i]] = ratios_[i];
  }
}
```

Figure 6.8

Moving on to the `DoBuild()` function. This is where the action happens and we apply a survival rate to our categories.

```cpp
/**
 * Execute the process
 */
void SurvivalConstantRate::DoExecute() {
  LOG_FINEST() << "year: " << model_->current_year();

  // get the ratio to apply first
  unsigned time_step = model_->managers().time_step()->current_time_step();

  LOG_FINEST() << "Ratios.size() " << time_step_ratios_.size() << " : time_step: " << time_step << "; ratio: " << time_step_ratios_[time_st
  Double ratio = time_step_ratios_[time_step];

  StoreForReport("year", model_->current_year());

  unsigned i = 0;
  for (auto category : partition_) {
    Double s = s_[category->name_];

    unsigned j = 0;
    LOG_FINEST() << "category " << category->name_ << "; min_age: " << category->min_age_ << "; ratio: " << ratio;
    StoreForReport(category->name_ + " ratio", ratio);
    for (Double& data : category->data_) {
      data -= data * (1-exp(-selectivities_[i]->GetResult(category->min_age_ + j, category->age_length_)  * (s * ratio)));
      ++j;
    }

    ++i;
  }
}
```

Figure 6.9

You will notice another set of logging, which is also encouraged to add all throughout the execution process. The levels of logging are described in Section 5.1

Adding a unit test

Still left to do - Samik and Marco to fill in after completing some unit tests.

7. Merging changes form a forked repository to a master repository

Before you consider this you need to make sure the following has been checked otherwise, you will waste developers time and patience. In brief summary you have to check that the your new code doesn't break any of the auto-differentiation code, so you need to build all of those. You must check that you have not broken any unit-tests and that all the model runner models have passed.

1. `doBuild archive` this will build all the autodiff librarys.
2. `doBuild release betadiff`
3. `doBuild modelrunner`
4. `doBuild test`
5. Enter the directory where this executable lives (e.g. `BuildSystem\bin\windows\test`), and run the CASAL2 executable (`casal2 -r`, or simply `casal2`) to check all the unit-tests pass.

only once the above is done, should you consider the remaining section. This section describes how to merge changes from a forked repository to the master repository. This is under the assumptions that the contributor has followed the rules laid out in Section 5. The aim of the forked repository is to maintain code integrity. So we only want contributors who have tested their documented code to be included into the master. This section is the opposite to Section 3 (where we pulled changes from the master repository into our forked repository). Here we are merging changes from a forked repository into the master, which will be incorporated into the next publicly available compiled version of CASAL2. From this example we can see from Figure 7.1 that our forked repository has three commits (underlined in blue) that we want to incorporate into the master repository.

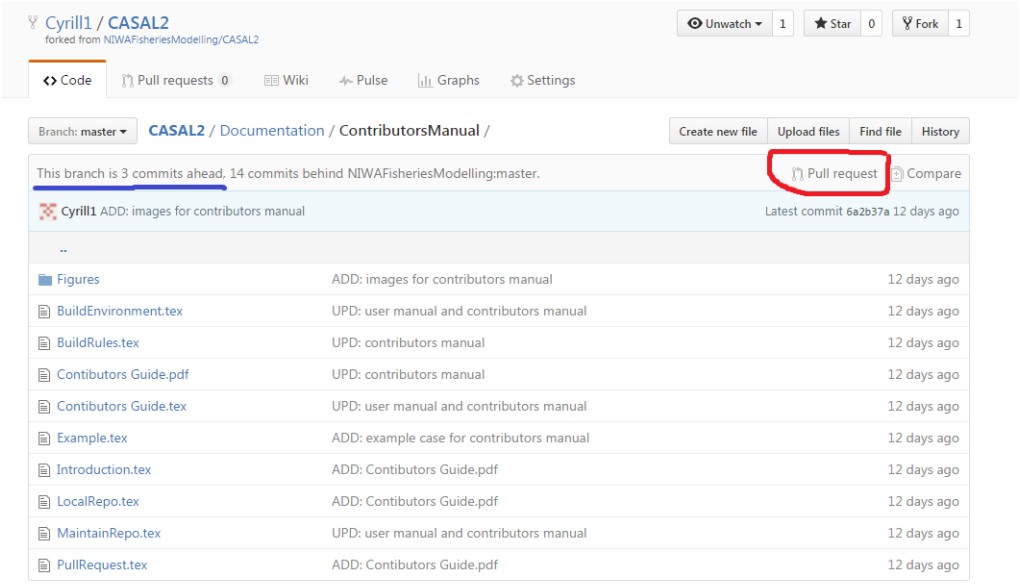

Figure 7.1

To incorporate these changes into the master you need to click on the "pull request" button circled in red. This will prompt a comparison of the changes that you are submitting for inclusion into the master, see Figure 7.2

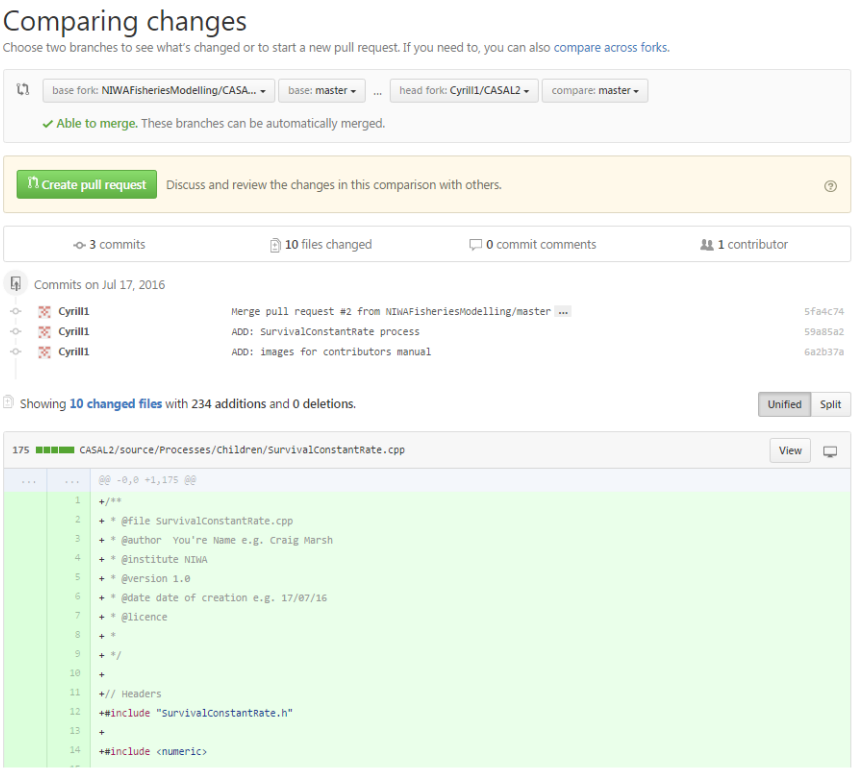

Figure 7.2

We can see that there are 10 files changed over three commits. A the top left corner there is a green button with "create pull request" click that button. This will open a pull request on the master repository as shown in Figure 7.3

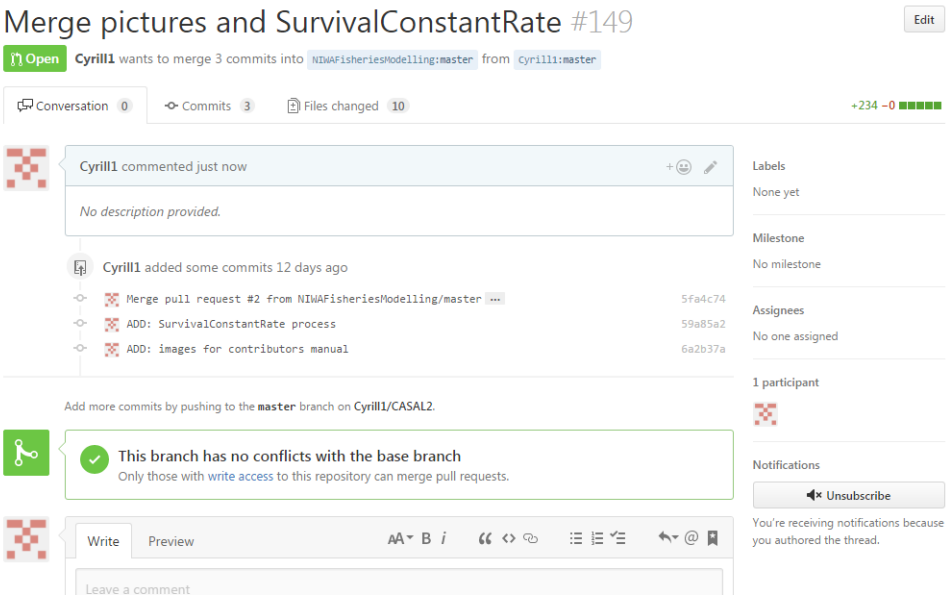

Figure 7.3

This will notify the maintainers of the master repository that a contributor is requesting a merge of the forked code into the master. Maintainers are able to look through the proposed code changed

and have conversations with the contributors if they need clarification. But at this point we are just waiting for the maintainers to accept the changes which then get incorporated into CASAL2 which then makes you a legend.

If you make it this far, we thank you for your development and contributions of this tool and we hope that you get great use out of it =)

8. Moderating Code for Maintainers

This section is for developer responsible for maintaining the github repository, specifically the best way to check a pull request. When we get a pull request the things you want to scrutinize are syntax and optimisation. Why I focus on these is because the automated build system test will tell you if the unit-tests are broken so don't waste your time looking at them. Hopefully who ever is requesting a pull-request has abided by the testing framework and this won't be an issue. **This section assumes the pull request passes all test's and builds**, please be diplomatic if this is not the truth remembering that we want to encourage collaboration on this project.

9. Adding Reports

This section gives some points when adding report classes to CASAL2. There are two variables that you must define when describing a report class, these are `model_state_` and `run_mode_`. I will try and explain what values you should give your report, but another avenue is to look at example report classes. The `model_state_` parameters tells the model when to print the report `State::kExecute` will call the `DoExecute` after each year and `State::kIterationComplete` will print the report once the model has run. So obviously if the report prints annual information or doesn't cache information you will want to use `State::kExecute` e.g. `Partition` else use `State::kIterationComplete`. **Important note** if you add a new report that has `model_state_ State::kExecute` you will need to tweak the R-library function `extract.mpd()`. At line 50 of the of `extract.mpd.R` there is the following line of code.

```
1 multi_year_reports = c("partition", "PartitionBiomass", "PartitionMeanWeight")
```

you will need to add the report label type to this vector so that R will process the report correctly.

The other command `run_mode_` just tells CASAL2 what run-mode should the report be generated in.