JEDI Documentation

Release 1

JEDI Core Team

Jun 05, 2019
Contents

1 Table of Contents
1.1 Background .................................................. 3
1.2 Working Practices ........................................... 4
1.3 Developer Tools and Practices .............................. 7
1.4 JEDI Environment ............................................ 33
1.5 Building, Testing, and Running JEDI ........................ 52
1.6 JEDI Components ............................................ 81
1.7 Background .................................................... 95
1.8 Working Practices ............................................ 95
1.9 Governance of a community system ....................... 95
1.10 Roles and Responsibilities .................................. 96
1.11 Conclusion .................................................... 96

2 Indices and tables ............................................... 99
Welcome to JEDI!

This documentation will help you get started with JEDI whether you are a user or a developer.
1.1 Background

1.1.1 JEDI High Level Requirements

As Earth observing systems are constantly evolving and new systems launched, continuous scientific developments for exploiting the full potential of the data are necessary. Data volumes are also increasing with time, requiring more and more efficient data assimilation. Given the cost of new observing systems, it is important that this process happens quickly. One goal of the new framework is to enable efficient research work and accelerated transition from research to operations.

As weather and environmental forecasting progress, they are also evolving towards a more comprehensive representation of the Earth system. The framework will have to cater to all components of the Earth system, enabling the evolution towards a fully coupled Earth system.

Data assimilation algorithms themselves are also evolving and progressing to better exploit available data. The framework will include existing operational DA algorithms and facilitate exploration of new DA science across domains and applications. It is important to note that a unified system does not mean a single configuration to be imposed on all partners as each agency can use or develop different applications within the framework.

The supercomputers where data assimilation systems are run are becoming more and more complex with always more and possibly heterogeneous processing elements. Using them efficiently is a growing concern in the community, with most centers exploring ways to improve scalability of their forecasting models and data assimilation systems. The framework will have to take scalability into account and facilitate adaptation to new architectures.

Requirements for operational and research uses are different. Generally, the system will have to be flexible for development and research, and easy for users to learn and test. At the same time, it is imperative that it satisfies operational requirements for robustness, efficiency, coding standards, and maintainability.

1.1.2 JEDI General Methodology

The requirements for JEDI are diverse and many aspects are very complex in themselves or together. However, over the last decade or two, software development technology has advanced significantly, making routine the use of
complex software in everyday life. Most recently, communication technology has also progressed to a level where working on the same project across the country has become common practice in the software industry. Together, these technologies make the goals set forward feasible.

The key concept in modern software development for complex systems is the separation of concerns. In a well-designed architecture, teams can develop different aspects in parallel without interfering with other teams’ work and without breaking the components they are not working on. Scientists can be more efficient focusing on their area of expertise without having to understand all aspects of the system. This is similar to the concept of modularity. However, modern techniques (such as Object Oriented programming) extend this concept and, just as importantly, enforce it automatically and uniformly throughout a code.

In order to facilitate collaborative work, modern software development tools have been and will continue to be used. These tools include version control, bug and feature development tracking, automated regression testing and provide utilities for exchanging this information. This is essential for working across agencies, possibly in different parts of the country, and will be used both for initial development and long term evolution and maintenance.

Having tools available, the first task will be to define interfaces between the components of the system. These interfaces will be generic and abstract. This task will benefit from the experience of other similar projects (e.g. OOPS at ECMWF). Based on these interfaces, high level model-agnostic applications will be progressively developed.

Once high level abstract interfaces are defined, existing codes will be progressively adapted to the new interfaces. Existing software will be modified to call the refactored parts to prevent divergence and maintain a continuously functioning system. A subset of interfaces that should be implemented first will be defined so that some applications can start using the new interfaces before everything is complete.

For operational applications, it must be ensured that the application of these principles will not adversely impact their ability to implement the code or negatively impact efficiency.

1.2 Working Practices

1.2.1 Branching and merging code

Following the git flow structure, repositories contain two special branches: the master branch which contains released versions of the code, and the develop branch which will contain all developments to be included in future releases. The branching model also provides three categories for other branches: feature branches where most developments happen, release branches for preparing new code releases and hotfix branches for bug fixes in already released code. This is different from older common working practices where master and develop are in the same branch (trunk) which is also where preparation of releases happen and where bugs are fixed.

A typical development will start by creating a feature branch (named feature/great_new_stuff) where the development work will happen. In most cases, this branch is created from the develop branch. Once the feature has been developed and tested, the new code is merged back in the develop branch.

As work happens in parallel, other developments might have been merged into the develop branch in the meantime. It is in principle the responsibility of the developer to first merge the develop branch into the feature branch and resolve conflicts. If the conflicts are simple to resolve this will be the case. If the conflicts are more complex to resolve, this is done in collaboration with the developer who introduced the conflicting change, or as a last recourse, by gatekeepers of the overall code.

For early detection of conflicts and easier resolution, modern practice recommend frequent merges of feature branches into the develop branch and frequent merges of the develop branch into the feature branch as work progresses. It is much more efficient to fix several small conflicts one after the other than large conflicts coming from several independent developments all at once after long periods of disconnect work. Early detection encourages discussions between developers to plan merging of future code changes before merges become too difficult.

Modern software design will also help in reducing conflicts in the development and merging process (explained below).
At some point, it will become necessary to prepare releases of the code. This process happens in specific release-branches. Once a release is ready and fully tested it is merged into both the master and develop branches and tagged. Typically, the tag will contain the release number (for example 2.0 for a major release, or 2.1 for a minor one). The master branch then contains the new official release and development work continues in the develop branch based on the release.

Despite all the care being taken in the testing, there will always be (hopefully rare) bugs needing fixing in any software. When such a bug is detected in a release, a hotfix branch is created from the master branch to implement and test the fix. Once a satisfactory fix is implemented, the bug fix branch is merged into both the master and develop branches and the release number is incremented (a new tag is created with a minor version number increased, for example from 2.1.0 to 2.1.1). For reproducibility, an existing tag should never be moved.

See the guide for [getting started with git-flow](#) for more details.

### 1.2.2 Forking and cloning repositories

GitHub and similar tools allow for cloning or forking repositories. Cloning is used by developers that have push permission to the official repository. In this case, the developers are asked to follow the git flow methodology. Forking is used by developers that do not have push permission to the official repository. Developers working in a forked repository are free to use the workflow of their choice, but we would recommend git-flow.

In the context of a unified data assimilation and forecasting system, we recommend the forking mechanism.

In practice, each developer will work on their own fork of the repository and clone from that work for editing code. Each developer can push to their own work freely (it is recommended to push often). Developers issue a pull request when contributions are ready to be merged in the central repository. It is possible for developers to pull branches from each other’s repositories and collaborate on a common feature. They can also share a branch on the central repository if the branch is pulled there by someone with sufficient permissions. This should be used for features that require interactions between more than a few developers.

In the collaborative context of a unified data assimilation and forecasting system, an advantage of the forking approach is that it can be used in a recursive manner. A central repository should be agreed between all the partners. From there, each partner can create a fork, preferably with automatic syncing. Within each partner organisation, each developer then forks their organisation’s fork as their central repository. For large partner organisations, it is even possible to add another level by team, group or division.

A common concern for operational codes is safety. In the proposed model, operational users can have their own fork of the repositories they require. For compiling the code, a clone of that repository is created on a local disk. If the fork and clone are synced manually nothing can happen until the operational user actively pulls new changes. Even if synced automatically, users can add their own tag or use a specific commit to prevent accidental changes. The operational clones can be located inside firewalls and on backed-up disks as required by operational guidelines. GitHub exists as a cloud based service but it is possible to install it on an organisation’s own server (GitHub Enterprise), thus the fork itself can be inside a protected network as well.

### 1.2.3 Reviewing code

We now discuss procedures to control the contents of branches, especially the develop and master branches. This is the role of the review process.

The functionality offered by GitHub and ZenHub will be used to increase the efficiency and reliability of the review process. The merging of branches should happen through a pull request within the GitHub environment. Each pull request should be assigned reviewers and the review should happen within the environment. Discussions and comments will be stored within the system and can be referred to later if needed. This is different from older common working practices where reviews happen by email and are almost impossible to trace later on.
The review process is an integral part of the development work and should be considered from the start. As a result, at the beginning of each development, it is assigned a ZenHub issue where discussion and documentation can happen during the development process and can be referred to during the review process and later if required.

Reviewers should be assigned for any new development as soon as it starts; assignments are done through ZenHub. Reviewers should be aware of what is coming ahead of time and not be surprised by unannounced massive changes. Reviewers can be reassigned so that the most relevant people are involved at any point in time. The number of reviewers is not fixed. For a trivial bug fix or very local changes that do not affect the rest of the system, one reviewer is enough. For more complex modifications, more reviewers should be included. It is suggested that a list of default potential reviewers is designated and maintained for each large components of the code (B matrix, observation operators, etc...) to assist in the process and that regular developers are also involved in reviewing each other’s code as an excellent way to promote a common culture and knowledge of the code.

It is in the interest of both developers and reviewers to review and merge code often in small incremental changes rather than massive changes at once. This facilitates the process, exposes code earlier to other developers who could be affected and facilitates the merging. In case of merge conflict, the developer should try to fix them first. If the conflict is complicated to resolve, the developer should liaise with and resolve the situation with the developer who has merged the conflicting changes. GitHubs and ZenHub provide the tools for that type of communication directly in the system. This will promote a common culture and knowledge of the code. Other developers or overall maintainers of the system can be brought into the discussion if it cannot be resolved by the two initial developers.

Once code has been reviewed and accepted in the review mechanism, it is merged as is without further modifications, clean-up or other improvements. Because the developers should have merged their branch up to the current level of the develop branch, the merge will be automatic and without conflict. Even gatekeepers should have their code reviewed.

Decisions in the review process could include scientific and technical aspects and it is possible to assign different reviewers for each aspect. However, in principle, discussions of a scientific nature should happen at the level of the ZenHub issue and before branches reach the level of the pull request.

An important aspect in the multiple level forking model is that review should happen at every level, thus giving several levels of control on the code. When developers create a pull request to the organisation’s repository, the code should be reviewed as described above. Then, another pull request will be issued from the organisation to the central repository, triggering another level of review. Depending on the level of the changes, the first or second review will be more or less important. In any case, discussions and documentation related to the feature (through the reviews and the ZenHub system) should be visible in both levels.

There is yet another chance to review changes when preparing a release. As any branch, a release branch can contain code modifications and should be reviewed. Git provides tools that help removing a given commit (merge or other) that is decided should not have been included. It is easier than it was in the past but it is not perfect and it should be the exception, not the norm. Reviews at every level should be treated seriously.

Finally, as it is good practice to merge often, the review process should be efficient. Modern tools help but reviewers should make code reviews a priority. As an example, some teams in the Agile/Scrum methodology do not allow developers to start work on a new issue as long as there are reviews pending. Although we cannot go that far, incentives to prioritize code reviews should be discussed.

Modern data assimilation and forecasting systems are very complex. As we enter an era when coupled system will become the norm, complexity will increase even more. It is important to recognise this and to accept that nobody can understand and control the whole system. This is why it is important that the reviewing of code is shared between people with different areas of expertise. It will also distribute the work and make the process more efficient.

1.2.4 Testing

Testing is a mandatory step of the development process and no branch should be merged without prior testing.

To assist developers and reviewers in their roles, the ecosystem includes facilities for testing the code. This entails two aspects: tests have to be developed and the environment should make it easy to add and run them.
Although some tests exist for various systems, it should be considered part of the development process to add tests to the existing collection. Developers are the best persons to write tests for their code and every major development should be included with its associated tests. The tests should be reviewed as any other part of the system and any by-passing or relaxing of a test criterion should be fully justified and documented.

An automated test system (e.g., Go CD, Travis) will be used. This system can be configured to automatically run a test suite on certain actions. For example, tests can run automatically for every push on a repository. Tests can also be run automatically every night or every weekend and can be configured to run on several platforms and with several compilers.

Running a full operational suite as a test is very expensive and also not the most helpful to prevent bugs from entering the system. A hierarchy of tests will be developed and provided, ranging from unit testing, regression testing, to low and high resolution application testing. Some tests will include code performance criteria to prevent unintentional or unnoticed creep of computational costs in the system. Scripts checking the conformance of the source code to a set of defined coding norms will also be included in the testing to facilitate the reviewing process.

The environment will be configured to run tests automatically and pull requests will be disabled until all tests in a predetermined suite pass without failure. For efficiency and cost reasons, this cannot include all levels of tests. Additional more expensive tests will be run regularly on the develop branch and issues reported back to developers as they are discovered. This ensures issues are detected early which will facilitate and accelerate the preparation of new releases to the master branch.

Although tests will run automatically on certain actions or at predetermined times, developers can run them manually at any time on their branch. They will be encouraged to do this regularly to detect potential problems early and fix them early.

### 1.2.5 Creating documentation

For writing guides and manuals, we are using Sphinx which is a Python package.

[Click here for tips on getting started with Sphinx](#)

Doxygen will be used for automatically generating documentation describing our code, such as inheritance diagrams, man pages and call trees.

We have created a GitHub repository for holding documentation called JCSDA/jedi-docs. Please place your documentation in this repository and place the appropriate links and text to your documentation in the top level index.html file.

### 1.3 Developer Tools and Practices

A certain amount of consistency is required on the part of the developers so that we can maintain efficient software development. Therefore, we have adopted several methodologies and tools that we ask everyone to use.

#### 1.3.1 Homebrew (Mac only)

If you use have an Apple computer running Mac OS X we highly recommend that you install Homebrew. This will make it much easier to install a number of other software tools that are indispensable for JEDI developers, including vagrant, git-flow, doxygen, and many more.

To install homebrew, copy and paste this into the command line of a bash shell:

```
/usr/bin/ruby -e "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/master/install)"
```
The installation of some JEDI tools such as Vagrant require new Homebrew features such as cask. To make sure you are using the latest version of Homebrew, type:

```
brew update
```

This updates the Homebrew application itself. To update all the packages that have been installed by Homebrew, type:

```
brew upgrade
```

### 1.3.2 Git flow

The [git flow primer](#) describes how to use native git commands to implement the flow. Since the writing of the primer, several modules have been created that provide git extensions that bundle native git commands into much simpler commands for each of the steps in git flow.

The cheat sheet below includes instructions on how to install git flow, as well as how to use git flow.

[Click here for a git flow cheat sheet](#)

The primer and cheat sheet both assume that you have permission to push commits into the develop branches of the remote GitHub repository. The typical case will be that in which the developer does not have such push permission, and instead must issue pull requests to get their code merged into the develop branch. The following steps are included to describe how to use git flow when you will be issuing pull requests (instead of directly pushing commits into the GitHub develop branch).

#### Installing git flow

This only needs to be done once on each machine. Following the instructions on the cheat sheet for the Mac:

```
brew install git-flow-avh
```

#### Initializing your local repository

These steps only need to be done once when you start a local repository.

```
git clone <path_to_remote_repository> # create local repository
git checkout --track develop # checkout and track develop branch
git flow init -d # initializes git flow with default values
```

It is possible to initialize git flow with different values but using all default values is very strongly recommended.

#### Adding a feature

Implementing a planned change is called “adding a feature” in the git flow terminology. This is a common operating mode and would include improvements, new features, and non-emergency defect repair.

**The idea is to repeat the following sequence for each feature:**

1. Make the change in your local repository in a new “feature” branch.
2. Push the new feature branch to the remote GitHub repository.
3. **Issue a pull request on GitHub to merge the new feature branch into the develop branch in the remote repository.**
   
   The owner of the GitHub repository will review and merge in your new feature branch.

4. **After your new feature branch is merged in, sync up with the remote GitHub repository.**

   In git flow, the “feature” command is used to assist with this process. Let’s say you want to call your new feature “perf-enhance” since you are working on performance enhancements.

   ```bash
   git flow feature start perf-enhance  # a new branch called feature/perf-enhance
       # is created and checked out
   # make edits, test code, etc.
   git flow feature publish perf-enhance  # this pushes your new branch
       # onto the GitHub remote repo
   ``

   On GitHub switch to your new branch feature/perf-enhance, and issue a pull request by hitting the “pull request” button. When the pull request screen comes up, make sure that you have your “feature/perf-enhance” branch designated as the “compare” branch and “develop” designated as the “base” branch.

   Click here to see details for creating a pull request on GitHub

   The owner of the GitHub repository will work with you to review and make any adjustments necessary as part of the process of accepting your changes. Once approved, the owner will merge in your “feature/perf-enhance” branch into the “develop” branch in the GitHub repository. Note that since “feature/perf-enhance” on the remote repository is no longer needed (it has been merged into the “develop” branch), it will be deleted in the remote repository (but not in your local repository).

   Once the merge on the remote GitHub repository has occurred, you need to get your local repository back in sync with the remote repository. This can be done by running the following:

   ```bash
   git remote update -p  # This synchronizes the metadata describing the changes that
   # have been done on the remote repository. The -p option "prunes"
   # branches that have been deleted on the remote repository which will
   # include your "feature/perf-enhance" branch.
   git checkout develop  # Switch to the develop branch in the local repository
   git pull origin develop  # Sync up the local repository with changes in the remote
   # repository (which will include your feature/perf-enhance changes).
   git branch -d feature/perf-enhance  # Remove feature/perf-enhance branch from your
   # local repository. Don't need the feature/perf-
   # enhance branch anymore since those changes are included
   # in the develop branch.
   ``

**Staying in sync with the remote GitHub repository**

All of the work to add in new features is done on the develop branch in the git flow methodology. Since there will be multiple people contributing to the develop branch, it is a good idea to sync up often to the develop branch (of the remote GitHub repository). A reason for doing this is to make sure that changes other people make are compatible
with the code you are developing (and vice versa). One way to get into the habit is to sync up every morning before getting started on your work.

Let’s say you are midway through the work on your feature/perf-enhance branch and you decide it’s a good time to sync up with the GitHub develop branch.

```plaintext
# Switch to the develop branch in your local repository.
git checkout develop
# Bring in the changes, if any, from the remote GitHub repository.
git pull origin develop
# Go back to the local feature/perf-enhance branch.
git checkout feature/perf-enhance
# Merge in the changes that were just pulled into the local develop branch.
# Note that this command is not necessary if the pull command above did not modify the develop branch.
git merge develop
```

**bugfix and hotfix branches**

Feature branches are intended for exactly that - new features or enhancements of existing code. If instead you want to fix a known bug in some branch of the repository, you should create a **bugfix** or **hotfix** branch.

The difference between **bugfix** and **hotfix** has to do with where they fit into the **git flow workflow**:

- **bugfix**: branches off of the **develop** branch or a specified **feature** branch
- **hotfix**: branches off of the **master** branch

These branches are created and finalized **as described above** for feature branches, e.g.:

```plaintext
# branches off of develop
git flow bugfix start wrongoutput
# branches off of master
git flow hotfix start wrongoutput
```

The default base for a bugfix branch is develop but you can also fix a bug in a feature branch as follows.

```plaintext
# branches off of myfeature
git flow bugfix start wrongoutput feature/myfeature
```

Bugfix and hotfix branches can be published and finalized **as described above** for feature branches, for example:

```plaintext
git flow bugfix publish wrongoutput
```

Once your branch is on GitHub, you can issue a pull request to merge it in to the relevant branch (master, develop, or feature). Our standard workflow is to delete the bugfix or hotfix branch on GitHub after it has been merged by an appropriate JEDI master.

Once it is successfully integrated into the desired branch, you may wish to delete your local branch manually using the standard git command:

```plaintext
git branch -d bugfix/wrongoutput
```

And/or, you can run this command periodically which will remove (−p is for **prune**) any of your local branches that no longer exist on GitHub:

```plaintext
```
1.3.3 Git-LFS

Git-LFS is a service to manage large files on GitHub repositories. LFS stands for Large File Storage.

Standard GitHub has a strict limit of 100 MB per file. Space limits for GitHub repositories are a little more fuzzy but you might get some polite complaints from user services if your repos exceed 1GB.

At JEDI, we use Git-LFS to get around these limits. Git-LFS allows for individual files of up to 2 GB and provides storage in units of 50-GB data packs (each with 50GB/month bandwidth), for a monthly fee.

Git-LFS works by storing large files on a remote server (called a Store) and then including text pointers to these files in the appropriate GitHub repositories in place of the files themselves. When the files are needed to run or compile your code, they are retrieved from the Store on an as-needed basis.

The logistics of how all this works will become more clear as you go through the process of installing and using Git-LFS.

Installing git-lfs

Git-LFS is a service but it is also an application. The application is a command-line extension to git that enables you to access and use the service.

If you are working from within the JEDI Charliecloud or Singularity container then you can skip this section; the git-lfs application is already installed as part of the JEDI environment.

If you have a Mac, you can install the git-lfs application outside the container using Homebrew:

```bash
brew install git-lfs
```

If you have a Windows or a linux machine or if you use MacPorts instead of Homebrew, you can go download and install the application from GitHub’s git-lfs site.

Using git-lfs

The first step in using Git-LFS is to identify files that are too large to be included in a GitHub repository. This certainly includes any files over 100 MB but it may also include smaller files that contain observational data sets or restart files for models - anything that is not code and that occupies a significant chunk of memory is a candidate for Git-LFS. Remember that GitHub prefers it if repos occupy no more than 1GB. So, even 10 MB files can push the limit if you have dozens of them.

If you plan to add any files to a JEDI repository that meet this description, then you should consult the section below on

Adding large files to a JEDI repository

If you are only modifying code then you should not have to worry too much about Git-LFS. Once it is set up for a given repository, the rest is pretty transparent to the user. Any files that are tracked by git-lfs will not occupy memory in your GitHub repository but you’ll still see them listed there just like the other files. If you select one of those files you may notice an inconspicuous message like this:
Otherwise everything should look pretty much the same. If it is a binary file you may see a polite message to the user that *we can’t show files that are this big right now*. However, you’d see the same message even if the file were not on LFS. In either case, if you select View Raw or Download, then GitHub will download the file to your computer, retrieving it from the LFS Store if necessary. If it’s an image file, GitHub will just retrieve it from the LFS Store and display it for you to behold.

When you pull or fetch and checkout a particular commit, then git LFS will check to see if you already have the tracked files on your system. This is called your **local LFS cache**. If so, it will replace the pointers with your local files. If not, it will download the LFS-tracked files from the remote Store.

An advantage of git LFS (in addition to the larger files sizes and more memory per repo) is that it takes less time to push, pull, and fetch repositories. If the large files don’t change, they can just sit on your local computer (in your local LFS cache) and there is no need to shunt them back and forth to and from GitHub.

Once Git LFS is installed, you can manage a LFS-enabled repo as you would any other; the standard fetch, pull, push, and clone commands work as before (beware of outdated documentation in google searches that says otherwise).

To see if Git LFS is enabled for a given repository and, if so, which files it tracks, go to the repository and enter

```bash
git lfs track
```

This will show you the tracked patterns. To see the actual files that are currently being tracked (i.e. stored on the LFS store instead of GitHub proper), enter

```bash
git lfs ls-files
```

For a complete list of git-lfs commands, enter

```bash
git lfs help
```

Or, to get more detailed information on any particular command, enter

```bash
git lfs help <command>
```

For technical details on how the pointers are implemented see the official Git LFS Specification. And, for details on how to access the Git LFS Store directly, see the Git LFS API.

For further documentation and usage tips, see GitHub’s help page and this tutorial.

### Adding large files to a JEDI repository

If you’d like to add one or more large files to an existing JEDI repository, it’s likely that that repository is already set up to use Git LFS. To see if this is the case, then go to the repository in question and enter

```bash
git lfs track
```

If this command does not return anything then Git LFS is not yet implemented for this repository. If that is the case, see **Activating Git-LFS for a JEDI repository** below.

More likely, the above command _will_ return a list of the files in this repository that are currently being tracked by Git LFS. For example:
Listing tracked patterns

test/testinput/*.nc (.gitattributes)
test/testinput/*.nc4 (.gitattributes)

This tells you that Git LFS is tracking all netCDF files with the extension .nc or .nc4 in the subdirectory test/testinput (all paths are relative to the top directory of the repo).

If the file or files that you wish to add to the repository are already covered by these tracked patterns, then you are done. There is nothing more you need to do. For example, if the tracked patterns were as listed above and if I were to add a file called newfile.nc to the test/testinput directory, then this new file would be tracked by Git LFS. If I then proceeded to commit this branch and push it to GitHub, newfile.nc would be copied to the Git LFS Store and a pointer to it would be generated and stored on GitHub.

If the current LFS tracked patterns do not match the new or modified files you wish to add, then you need to define new patterns that do match. You do this with the `git lfs track` command, for example:

```
git lfs track "*.nc"
git lfs track "Documentation/*.ps"
git lfs track "mydata/**"
```

The `git lfs track` command accepts full directories or wildcards as shown above. The first command tells get-lfs to track .nc files anywhere in the directory tree of the repository. The double asterisk in the third command instructs git-lfs to recursively include all subdirectories. Paths are relative to the top level of the repository. You can specify as many patterns as you wish. These will all be stored in the .gitattributes file in the top directory of the repo and can be listed with the `git lfs track` command as described above (omitting arguments will generate the list).

Once you have your tracking patterns set up, then you can proceed to add your files.

**Note:** Be sure to set up the appropriate tracking patterns before you add your large files to the repository.

So, you’re ready to go. However, it is worth emphasizing that space is limited even on the LFS store and it’s not easy to remove a file once it is there (only the repo administrator can do this). Without careful attention, the accumulation of large files can add up quickly. So, please be prudent when adding large files to a JEDI repository.

**Note:**

**Before adding large files to a JEDI repository, please ask yourself these questions:**

- Will these files be useful to the JEDI community?
- Am I only including the files/data necessary to run a particular test or demonstration (pruning out all unnecessary auxiliary files/data)?
- Will these files remain useful indefinitely, without the need for frequent updating?

**Activativing Git-LFS for a JEDI repository**

Most JEDI users and developers can safely skip this section. By the time you read this, most relevant JEDI repositories will have already been configured to use Git-LFS.

As described above, to see if the repository is already set up to use Git LFS, go to the repository and enter `git lfs track`...
If this generates a listing of tracked patterns then you can skip this section; Git-LFS is already set up.

If it does not, then you can activate Git LFS by going to the top directory of that repository and entering

```
git lfs install
```

This will activate git-lfs for that repository.

Now you have to tell git which files you want to store on the remote LFS Store. You do this with the git-lfs track command as described above, for example:

```
git lfs track "*.nc"
git lfs track "test/testinput/*.nc4"
```

Entering one or more of these commands will create (or append) a .gitattributes file in the top level of your repository where your specifications will be stored. So, in order to save your LFS specifications for posterity, you should tell git to track this file:

```
git add .gitattributes
```

Now you can add your large files and the next time you commit and push this branch to GitHub, it will be properly configured for Git LFS.

**IMPORTANT:** Existing files that satisfy your pattern specifications will not be moved to the LFS Store. The reason for this is that they are already part of your GitHub history. They exist in previous commits so they already occupy memory on GitHub. Replacing them with pointers would be pointless, so to speak, because it would not save any memory. Even if you were to delete those files from your repository, re-commit, and then re-commit again after adding them back in, GitHub is smart enough to know that these are the same files that were there before so it will use the versions it already has in memory (unless you change the file names or the files themselves).

If you are really courageous and determined, there is a way to move existing files to LFS. This would require you to first delete the files from the repo history (make sure you move the files someplace safe first!). Then you can run `git lfs install` and `git lfs track` as described above and then move the files back to the repo. Then when you commit and push to GitHub, the files will be stored on Git-LFS. Another way to move existing files to the LFS Store is with the `git migrate` command.

However, you can only do this if you have push permission to the repository. If you do have push permission please use this with **great caution** because it does (literally) rewrite history! Pre-LFS versions of the repo may fail tests that they previously passed.

It is much better to:

**Note:** Use Git-LFS right from the beginning when you add large files to a JEDI repository

### 1.3.4 Sphinx

Sphinx is a Python package that can be used to create documentation in various formats that include HTML, LaTex and man pages. For input Sphinx uses reStructuredText which is a variety of markdown language. Markdown is a simple, easy to use textual representation of a complex markup language such as HTML.

Click here for details on Sphinx

**Installing Sphinx**

This step only needs to be done once for each repository when starting sphinx. Note that the JCSDA/jedi-docs repository has already had this step run.
Assuming that you have Python installed, do the following to install Sphinx:

```
pip install -U sphinx  # for Python 2
pip3 install -U sphinx  # for Python 3
```

### Initial Configuration for Using Sphinx

To set up for using sphinx in a repository:

```
cd my-repo
mkdir docs  # using the name "docs" will allow ReadTheDocs to find and process your
  →files
cd docs
sphinx-quickstart  # Answer queries as shown below
  →which
  > Separate source and build directories (y/n) [n]: # hit return for default
  →brackets
  > Name prefix for templates and static dir [ ]: # is shown in square
  →brackets
  > Project name: MyProject
  > Author name(s): Your Name # spaces are okay
  > Project release [ ]: 1.0.0 # software version number
  > Project language [en]: # hit return for default
  →(English)
  > Source file suffix [ .rst ]:
  > Name of your master document (without suffix) [index ]:
  > Do you want to use the epub builder (y/n) [n ]:
  > autodoc: automatically insert docstrings from modules (y/n) [n ]:
  > doctest: automatically test code snippets in doctest blocks (y/n) [n ]:
  > intersphinx: link between Sphinx documentation of different projects (y/n) [n ]:
  > todo: write "todo" entries that can be shown or hidden on build (y/n) [n ]:
  > coverage: checks for documentation coverage (y/n) [n ]:
  > imgmath: include math, rendered as PNG or SVG images (y/n) [n ]:
  > mathjax: include math, rendered in the browser by MathJax (y/n) [n ]:
  > ifconfig: conditional inclusion of content based on config values (y/n) [n ]:
  > viewcode: include links to the source code of documented Python objects (y/n) [n ]:
  > githubpages: create . nojekyll file to publish the document on GitHub pages (y/n)
  →[n ]: y
  > Create Makefile? (y/n) [y ]:
  > Create Windows command file? (y/n) [y ]: n
```

This creates a configuration file (conf.py, which can be subsequently edited), a Makefile for creating your document in different formats (e.g., HTML) and an initial index.rst file. Also, directories are created for holding the output of make (_build), custom HTML templates (_templates) and custom stylesheets (_static).

Sphinx has different “themes” which set the style of your html pages. ReadTheDocs can pick up on these themes and if the theme is called “default”, then ReadTheDocs will substitute its own page style. However, sphinx-quickstart writes the conf.py file with a theme called “alabaster”. To change the theme to default (which looks nice in both Sphinx and ReadTheDocs), do the following:

```
vi conf.py  # substitute your favorite editor
  →find the line: html_theme = 'alabaster'
  →change 'alabaster' to 'default'
```

---

1.3. Developer Tools and Practices
Writing Your Document

Take a look in the index.rst file that was created by sphinx-quickstart. It has a table of contents, specified by the “toctree” directive. Whenever you add another .rst file, place a reference to that file in the table of contents. Note that the string you entered for your project’s name appears in several places in the index.rst file. At the bottom, note the creation of an index (“genindex” directive). Entries in the index are created wherever you place an “index” directive in the text of your .rst files.

reStructuredText is easy to use, yet it has an extensive set of features. Probably the best way to get going is to look up examples on the web. Also, the sphinx website has a great primer for reStructuredText which can be viewed by clicking the link below.

Details on reStructuredText

Once you are ready to build your documentation, run:

```
cd my-repo/docs  # the directory you were in when you ran sphinx-quickstart
make html        # create web pages
make latex        # create a LaTex manual
make latexpdf     # create pdf from the LaTex files
make man          # create man pages
```

After running make, the output will appear in the _build directory in a subdirectory corresponding to the output format you selected (e.g., _build/html for the output of “make html”).

HTML pages can be viewed using the URL file form. If you built your HTML in the directory

```
/users/me/my-repo/docs/_build/html
```

then use the following URL to view your pages

```
file://users/me/my-repo/docs/_build/html/index.html
```

More Help with Getting Started

See the following link for more details on building documents with sphinx:

Details on document building

1.3.5 Doxygen

We at JEDI use Doxygen for generating man pages, inheritance diagrams, call trees and other types of html documentation that is linked to specific blocks of source code such as classes, functions, and subroutines. For generating web-based manuals, guides, and tutorials we use Sphinx.

Doxygen is open-source software that was developed by Dimitri van Heesch and is distributed under the GNU General Public License. For further information on the project see the Doxygen home page and for extensive documentation on how to use it see:

The Doxygen User Manual

In what follows we give practical tips on how to use Doxygen within the context of JEDI.

Note: The most important part of this document are the instructions on how to add Doxygen documentation to C++ and Fortran code
Don’t worry about the details of how to use Doxywizard. This is optional. Doxygen configuration files are included with most JEDI bundles and if you want to see the output you can just enable the Doxygen build when you build the bundle and view output with a normal web browser.

Also, as the project proceeds, the JEDI team will provide web pages where you can view the Doxygen html output for current develop branches and prior releases. Stay tuned to this site for further details.

**All we ask is that you document any code that you add.**

### Installing Doxygen

Doxygen is included in the JEDI CharlieCloud and Singularity containers and may already be installed on your system. To check whether it is already installed in your environment, just type this at the command line:

```
doxygen --help
```

If it is not already installed, you can obtain executable binary files for Mac OS X, Linux, and Windows through the Doxygen web page or you can download the source code from GitHub and build it yourself.

Alternatively, if you have a Mac, you can install Doxygen with **Homebrew**

```
brew install doxygen # (Mac only)
```

Depending on how you install Doxygen, you may be prompted for optional add-ons, including Doxywizard and Graphviz. We recommend that you say yes to both. Doxywizard is a convenient Graphical User Interface (GUI) for configuring and running Doxygen and Graphviz is a plotting package that will enable you to generate inheritance diagrams and call trees.

In particular, Graphviz includes an interpreter for the DOT graphical display language. A dot interpreter might already be installed on your system. For example, if you installed doxygen via Homebrew or if you use the JEDI Charliecloud or Singularity container, you may not need to install anything else. To check, try running:

```
dot --help
```

If it’s not already there you can install Graphviz using the executable binaries available from their download site or you can install it explicitly with **Homebrew**:

```
brew install graphviz # (Mac only)
```

### Documenting C++ source code

There are several ways to include Doxygen documentation in C++ source files. We recommend the Qt style, as illustrated in this example:

```c++
// ----------------------------------------------------------------------------
/*! \brief Example function */
/*! \details **myfunction()** takes a and b as arguments and miraculously creates c. I could add many more details here if I chose to do so. I can even make a list:
 * * item 1
 * * item 2
 * * item 3
 * 
 * \param[in] a this is one input parameter
 * \param[in] b this is another
 * \param[out] c and this is the output
```

(continues on next page)
Since these directives are located within comment blocks, they do not affect the compilation of the code.

A few things to note. First, the documentation for a function or class comes in a Doxygen comment block immediately before the function or class is defined. The Doxygen block begins with /*! and ends with */. Each line in between begins with *. Doxygen commands are indicated with @ or, alternatively, \. `\brief` gives a brief description that will appear in html and other lists whereas `\details` gives further details as would appear in a man page. `\param` describes the arguments of the function while multiple `\author` and `\date` items can provide a history, tracking the function’s development. `\warning` provides useful usage tips to the user or developer.

These are only the essentials; there are many more...

Doxygen commands

documented in the online manual.

Note also that Doxygen supports Markdown language features for further formatting of the output. Examples of Markdown above include the asterisks in **myfunction()** (bold type) and the bulleted list.

Doxygen also supports latex for including formulae in latex and html output. Latex math mode is delimited by \f$ symbols as follows:

```
/*!
 * This is an equation: \f$\nu = \sqrt{y_2}\f$
 */
```

Note - if you are configuring doxygen yourself, you must enable the USE_MATHJAX option in order for latex formulae to compile. If you are using the default Doxyfile provided with the repository, there is no need for any action on your part - Mathjax is already enabled.

**Documenting Fortran source code**

Including Doxygen documentation in Fortran is similar to C++ as described above, but with appropriate Fortran comment indicators. Also, the Doxygen parameter descriptions can follow the argument declarations as demonstrated here:

```
! --------------------------------------------------------------------------------------
! \brief Example function
!!
!! **myfunction()** takes a and b as arguments and miraculously creates c.
!! I could add many more details here if I chose to do so. I can even make a list:
!! * item 1
!! * item 2
!! * item 3
!!
!! \author L. Skywalker (JCSDA)
!!```

(continues on next page)
!! \date A long, long, time ago: Created
!!
!! \warning This isn't a real function!
!!

subroutine myfunction(a, b, c)
  integer, intent(in) :: a !< this is one input parameter
  integer, intent(in) :: b !< this is another
  real(kind=kind_rea), intent(out) :: c !< and this is the output
[...]

The Doxygen code block here begins with \>', and subsequent lines begin with <!--. The parameter definitions begin with <!--. The supported Doxygen commands are the same as in C++.

Running Doxygen and Viewing the Results

You may never need to run Doxygen yourself. As noted above, the JEDI team plans to provide Doxygen-generated html output on public web sites for specific JEDI releases and for the current develop branches. This is still in preparation.

But, if you have added Doxygen documentation to a feature branch that you are working on, you may want to see how it looks before doing a pull request. This is straightforward to do.

If you are working with a particular JEDI bundle, then it is likely that this bundle is equipped to build the Doxygen documentation. Just edit the CMakelists.txt file in the top level of the bundle repository (e.g. ufo-bundle) and look for a code snippet that resembles this:

```makefile
# Build Doxygen documentation
option(BUILD_UFO_BUNDLE_DOC "Build documentation" ON)
```

Just make sure this is set to ON.

Then, proceed to build jedi as normal, running ecbuild and make from a build directory <build-dir> (this should be different than the location of the source code). The Doxygen html output will then be located in a directory called <build-dir>/Documentation/html. Just load any of the html files in this directory into your browser and navigate the the Main Page using the menu at the top.

You can also run Doxygen manually, as follows

```
mkdir -p <build-dir>/Documentation
cd <build-dir>/Documentation
ecbuild <path-to-config-file>
doxgen
```

Then, as with the automated bundle build, the Doxygen-generated html output will be located in the directory <build-dir>/Documentation/html and you can view it with a web browser by loading any of the html documents that you see there.

Note that these manual instructions are specifically for JEDI repositories. In this case, the <path-to-config-file> should point to a directory that includes a file called Doxyfile.in. Examples include the Documentation subdirectories in the ufo-bundle, fx3-bundle, or oops repositories. The ecbuild step above converts this into a Doxyfile with the proper path information.

Alternatively, If you create your own Doxyfile with Doxywizard or with doxygen -g, then you can skip the ecbuild step and just run the doxygen command from the same directory as the Doxyfile (you could also specify the configuration file explicitly with the -g option to doxygen).
Or, you can generate the html output and view it using the Run Doxygen and Show HTML Output buttons on the Doxywizard GUI.

The JEDI source code already has some Doxygen documentation within it. So, even before you add your own documentation, you can run Doxygen on a particular JEDI repo and view the results. We currently use Doxygen to generate html files but it can also be configured to produce man pages and latex output.

Note: If you use a custom configuration file generated by Doxywizard or some other means, then the output will be located in whatever directory is specified by the OUTPUT_DIRECTORY declaration in the Doxyfile. This may be different than as described here.

After you load some html document from the Doxygen tree into your web browser, then you can use the menus to peruse the files, functions, namespaces, classes, etc. Selecting Classes-Class Heirachy will give you an inheritance diagram like this:

Selecting a file from the File List will let you see the documentation for the functions and classes it contains, including call diagrams. Here is an example of doxygen-generated documentation for a function - select the image for a closer look (note that most JEDI functions do not yet have this level of Doxygen documentation).
This is only the beginning - we encourage you to dive in and explore! For further details on what you find, consult the Doxygen User Manual.

**Doxywizard and Customizing the Doxygen Build**

Most JEDI repositories contain a Doxyfile configuration file (typically in the docs subdirectory) so there is no need for you to create a new one. Still, there may be situations in which you’d like to change look or content of the Doxygen documentation. You can either do this by editing the Doxyfile directly or using by using Doxywizard.

As mentioned above, Doxywizard is a convenient Graphical User Interface (GUI) for configuring and running Doxygen. It’s often installed together with doxygen as an optional extension.

To configure and run Doxygen with Doxywizard, just start up the application and begin filling in the menu items as shown here:
Take note in particular of **Step 1** at the top, namely specifying the directory from which Doxygen will run. If you select **Save** when you exit Doxywizard, Doxygen will create a configuration file in this directory called **Doxyfile** that you can later load into Doxywizard (via the File-Open... menu item) or edit manually. Then specify the source code directory and the destination directory (the project name is optional).

**Tip** Be sure you select the **Scan recursively** option when specifying the directory for the source code.

**Tip** We recommend that you place the Doxygen output in a directory outside of the JEDI repositories. If you do select an output directory within the JEDI repos, please exclude it from your commits so your files are not uploaded to the main JEDI repos on GitHub.

After you finish filling in this Project page, select **Mode** from the Topics menu on the left. Here make sure you select **All Entries** and **Include cross-referenced source code in the output**. Also, you may wish to optimize for either C++ or Fortran output.
Then proceed to the **Output** menu item on the left and make sure **html** is selected. Then select **Diagrams** and, if you installed GraphViz as described *above*, select **use dot tool from the GraphViz package**. And, select the diagrams that you'd like dot to generate:

There is one more thing you may need to do in order to get dot to work correctly. Select the **Expert** menu item at the top of the window (between *Wizard* and *Run*) and scroll down the menu on the left to select **dot**. First make sure the **HAVE_DOT** item is checked and then scroll down to specify the **dot path**, which is likely `/usr/local/bin/dot`. 

That is sufficient to run Doxygen but you may wish to browse some of the other items on the **Expert** menu, particularly under **Build**. When you’re finished, select **Run** from the top menu to get to the run screen and then select the **Run doxygen** button on the upper left to run Doxygen.
Wait patiently for it to run - it may take a few tens of seconds, particularly if you asked to generate many graphs.

If you’d rather not use the Doxywizard GUI, you can do all of the above and more by creating the Doxyfile configuration file manually from the command line and then editing it directly to select the options you want. To manually generate a Doxyfile, go to your directory of choice and type:

```bash
doxygen -g
```

Then, after editing the file to specify your configuration options (including the source and output directories), just type this thereafter (from the directory that contains the Doxyfile):

```bash
doxygen
```

To see the glorious abundance of configuration options, consult the Doxygen Manual.

If you have any problems, try consulting the Troubleshooting section of the Doxygen manual or the Doxygen tag on Stack Overflow - or email Mark (miesch@ucar.edu) or Steve (stepenh@ucar.edu).
To view the output as a man page, first make sure you have enabled the `GENERATE_MAN` option by selecting it in the "Expert-Man" menu of Doxywizard or by editing the Doxygen file. Then navigate to the `man/man3` subdirectory of the output directory. There you can type `ls` to see what man pages are available to view. These include files, namespaces, directories, and classes. To view one, type e.g.

```
man ./qg_fields.3
```

The `.3` extension (and the `man3` directory name) refers to section 3 of the `man` organizational structure, which is typically reserved for library functions. You can change this by changing the Doxygen variable `MAN_EXTENSION`.

In the future, we plan to maintain a central directory tree for the man pages that you will be able to include in your `MANPATH`, thus avoiding the `./` syntax above. But this is still under development.

### 1.3.6 CMake, CTest, and ecbuild

The JEDI manufacturing system (build, test and package) is based on the CMake tool suite. CMake is open source and its purpose is to facilitate the creation of make files that can be used to compile, test and install your software. For example, you can tell CMake that you need the netcdf and openmpi libraries, and it will automatically find those on your machine and place necessary paths, in the make files it generates, for compiling and linking your source code. CMake is in widespread use and the CMake website includes documentation and tutorials to help you get started.

CTest is the part of CMake that handles testing your code. CTest allows for an easy way to run your newly built programs with various argument and option settings, and then check the results against expected output. This system is well suited for short, fast running tests that tend to be in the unit level testing category. We may need to go beyond CTest to address our large scale system testing that is geared more toward performance benchmarking and verifying functionality on HPC systems.

Ecbuild is a set of CMake macros provided by the ECMWF that assist with the specification of the manufacturing processes. Along with ecbuild we are using two ECMWF libraries called eckit and fckit. Eckit is a C++ library that provides utilities including logging, MPI, configuration file (JSON, YAML) parsing and math functions. Fckit is a Fortran tool kit that provides similar utilities as eckit, plus helper functions to convert strings and arrays between Fortran and C/C++, and extending the unit test framework to Fortran.

#### CMake and CTest

The CMake developers provide a single package for CMake and CTest. Documentation and downloads are available at the [CMake website](https://cmake.org).

#### Installing CMake and CTest

This step is only necessary if you are working outside the JEDI Charliecloud or Singularity containers.

For the Mac, use `homebrew` to install CMake.

```
brew install cmake
```

For Windows and Linux systems, see the [CMake downloads website](https://cmake.org) for packages and instructions.

#### Using CMake and CTest

At the heart of CMake are the `CMakeLists.txt` files. These files are where the specification takes place for all manufacturing processes. This specification may include items such as where the source code lives, dependencies for that
source code, what libraries to link in, configuration of tests and the selection of programs and scripts for installation. Here is a CMake tutorial that is helpful for getting an idea of what goes into a CMakeLists.txt file.

Once the CMakeLists.txt for a project are created, all one does to build, test and install your software is:

```
cmake <dir_where_toplevel_CMakeLists.txt_file>  # generate the make files
make               # compile everything
ctest              # run the tests
make install       # install the programs and scripts
```

In the example above, the `<dir_where_toplevel_CMakeLists.txt_file>` argument to cmake is the Linux path to wherever the top level CMakeLists.txt file exists (only the directory part of the path). This path can be relative or absolute. Here are some examples for running cmake:

```
cmake .          # top level CMakeLists.txt is in the current directory
cmake ..         # top level CMakeLists.txt is one up from the current directory
cmake $HOME/projects/my-project # top level CMakeLists.txt file lives # in $HOME/project/my-project
```

Output from the cmake command is captured in the following files:

```
./CMakeFiles/CMakeOutput.log  # messages from cmake
./CMakeFiles/CMakeError.log   # errors and warnings from cmake
```

The ctest command without arguments will run the entire set of tests. In the case that you want to run a specific test or you want more information on tests that failed, you can run individual tests using ctest as shown below.

```
ctest -R test_ufo_radiosonde # this runs just the one test
ctest -R test_ufo_*          # file globbing and regular expression can be
                             # applied to select a subset of tests to run
ctest -V -R test_ufo_radiosonde # -V increases the verbosity of output
```

**Warning:** Many unit tests use MPI, which can require additional MPI configuration. For example, using OpenMPI on the Mac typically requires the following to enable oversubscribing (which means running more MPI processes than available cores). Note that extra MPI processes beyond the number of cores on a system do not actually run in parallel, but that’s okay with short, fast-running programs such as unit tests.

To enable oversubscribing on the Mac with OpenMPI:

1. Create the file: $HOME/openmpi/mca-params.conf
2. Place the following in the mca-params.conf file

```
# This Mac has 2 cores. Enable oversubscribe so that more than 2 MPI
# processes can be run on this system.
# This Mac has 2 cores. Enable oversubscribe so that more than 2 MPI
# processes can be run on this system.
rmaps_base_oversubscribe = 1
```

Test output is captured in the files:

```
./Testing/Temporary/LastTest.log  # output from the last invocation of ctest
./Testing/Temporary/LastTestsFailed.log  # names of the tests that failed during
                                        # the last invocation of ctest
```

1.3. Developer Tools and Practices
Note: It is highly recommended that you build your code in a directory that is separate from the directory where the source code lives. CMake does not restrict you to do this, but doing so will keep the source directories free from all of the clutter that the build process produces such as object files, the generated make files, and additional CMake configuration and log files. If you build in a separate directory, one simple remove command will clean up the entire build area (without danger of removing source files) and keep the source git repository clear of extra files that you do not want to check into the repository.

CMake provides many controls which are enabled through specifying the -D command line option. See the CMake variables documentation for details. This list is extensive, and probably the most relevant is CMAKE_INSTALL_PREFIX, which is used to specify where the programs and scripts are to be installed. By default, this is /usr/local. However, if you don’t have write permission to /usr/local, then you will need this control to be able to do the install step. Let’s say that you want to install in your home directory in the path $HOME/tools. Then run cmake as follows:

```
cmake -DCMAKE_INSTALL_PREFIX=$HOME/tools $HOME/projects/my-project
```

Another set of useful controls are those for setting which compilers will be used for building your project. CMake will search your system in common directories (/bin, /usr/bin, /usr/local/bin, etc.) for compilers and libraries needed by your project. It’s common for several versions of compilers to exist on a given machine and it’s not always clear which one CMake will choose. These controls can be used to force CMake to use the versions you want.

```
cmake -DCMAKE_C_COMPILER=/usr/local/bin/gcc $HOME/projects/my-project # C code
```

```
cmake -DCMAKE_CXX_COMPILER=/usr/local/bin/g++ $HOME/projects/my-project # C++ code
```

```
cmake -DCMAKE_Fortran_COMPILER=/usr/local/bin/gfortran $HOME/projects/my-project # Fortran code
```

# Note that combinations of these can be issued with one CMake command if you have a mix of source code languages. Say you've got C, C++ and Fortran.

```
CMP_ROOT=/usr/local/bin
cmake -DCMAKE_C_COMPILER=$CMP_ROOT/gcc -DCMAKE_CXX_COMPILER=$CMP_ROOT/g++ -DCMAKE_Fortran_COMPILER=$CMP_ROOT/gfortran $HOME/projects/my-project
```

CMake also has tools that are useful for debugging. In particular, the --trace and --debug-output options show every line of every script file that is executed while cmake is running.

**ecbuild**

The JEDI software stack links directly to the public ecbuild, eckit, and fckit GitHub repositories provided by ECMWF. In particular, public releases from these repositories have been cloned from GitHub, compiled, and included in the JEDI Singularity and Charliecloud containers.

Ecbuild does enforce the restriction recommended above on building your project outside of the source directories.

**Installing ecbuild**

As before, the steps shown in this section are only necessary if you are working outside the Singularity and Charliecloud containers.

For all systems, you need to have CMake, eigen3 installed before installing ecbuild. To install these on the Mac:
JEDI projects use Boost header-only libraries and building Boost is not required.

For Windows and Linux systems, see the CMake downloads website, Eigen website and Boost website for packages and instructions.

Since ecbuild is actually a collection of CMake macros there is no compiling required, thus no need to run make nor ctest. In the following example, the ecbuild clone is going to be placed in $HOME/projects and the build directory will be $HOME/projects/ecbuild/build.

```
# create the ecbuild clone and make sure you are on the master branch
cd $HOME/projects
git clone https://github.com/ecmwf/ecbuild.git
cd ecbuild
git checkout 2.9.4 # check out the most recent release

# create the build directory
mkdir build
cd build

# install ecbuild
rm -rf ..
# This assumes that you have write permission in /usr/local
sudo make install

# if you don’t have permission to write into /usr/local
make -DCMAKE_INSTALL_PREFIX=$HOME/tools ..
make install
```

Once ecbuild is installed, it can be used to build and install the eckit and fckit libraries. Currently, it is recommended to only install eckit, since fckit is generally built along with the JEDI code. This is done because the JEDI team often make changes to fckit and we generally work from our own fork.

For the following code example, assume that the clones are placed in $HOME/projects and the build directories are subdirectories of the clones called “build”.

```
# create the eckit clone
cd $HOME/projects
git clone https://github.com/ecmwf/eckit.git
cd eckit
git checkout 0.23.0 # check out the most recent public release

# create the build directory
mkdir build
cd build

# build, test, install eckit
#
# Note the use of ecbuild in place of cmake
#
# If no write permission in /usr/local, add -DCMAKE_INSTALL_PREFIX=$HOME/tools
# to the ecbuild command and omit the :code:`sudo` in the :code:`make install`
eckit
make
```

(continues on next page)
Using ecbuild

The ecbuild installation provides a command called ecbuild which is a direct replacement for the cmake command. Ecbuild simply loads its set of macros and then passes all appropriate arguments and options on through to a call to cmake. For example, you can use the option `-DCMAKE_INSTALL_PREFIX` with ecbuild and this gets passed through to cmake.

Ecbuild is the workhorse for building and testing (and eventually installing) the JEDI software. Once ecbuild and associated libraries (eigen3, eckit) are installed, all subsequent manufacturing is done using the ecbuild command in place of cmake. The output from ecbuild is captured in the file:

```
./ecbuild.log
```

Ecbuild has its own options which can be inspected by running `ecbuild --help`. Here is sample output:

```
>> ecbuild --help

USAGE:

  ecbuild [--help] [--version] [--toolchains]
  ecbuild [option...] [--] [cmake-argument...] <path-to-source>
  ecbuild [option...] [--] [cmake-argument...] <path-to-existing-build>

DESCRIPTION:

  ecbuild is a build system based on CMake, but providing a lot of macro's
  to make it easier to work with. Upon execution,
  the equivalent cmake command is printed.

  ecbuild/cmake must be called from an out-of-source build directory and
  forbids in-source builds.

SYNOPSIS:

  --help Display this help
  --version Display ecbuild version
  --toolchains Display list of pre-installed toolchains (see below)

Available values for "option":

  --cmakebin=<path>
    Set which cmake binary to use. Default is 'cmake'

  --prefix=<prefix>
    Set the install path to <prefix>.
    Equivalent to cmake argument "-DCMAKE_INSTALL_PREFIX=<prefix>"

  --build=<build-type>
    Set the build-type to <build-type>.
    Equivalent to cmake argument "-DCMAKE_BUILD_TYPE=<build-type>"
    <build-type> can be any of:
```

(continues on next page)
- debug : Lowest optimization level, useful for debugging
- release : Highest optimization level, for best performance
- bit : Highest optimization level while staying bit-reproducible
- ...others depending on project

--log=<log-level>
Set the ecbuild log-level
Equivalent to "-DCECBUILD_LOG_LEVEL=<log-level>"

<log-level> can be any of:
- DEBUG
- INFO
- WARN
- ERROR
- CRITICAL
- OFF

Every choice outputs also the log-levels listed below itself

--static
Build static libraries.
Equivalent to "-DCECBUILD_SHARED_LIBS=OFF"

--dynamic, --shared
Build dynamic libraries (usually the default).
Equivalent to "-DCECBUILD_SHARED_LIBS=ON"

--config=<config>
Configuration file using CMake syntax that gets included
Equivalent to cmake argument "-DCECBUILD_CONFIG=<config-file>"

--toolchain=<toolchain>
Use a platform specific toolchain, containing settings such as compilation flags, locations of commonly used dependencies.
<toolchain> can be the path to a custom toolchain file, or a pre-installed toolchain provided with ecbuild. For a list of pre-installed toolchains, run "ecbuild --toolchains".
Equivalent to cmake argument "-DCMAKE_TOOLCHAIN_FILE=<toolchain-file>"

--cache=<ecbuild-cache-file> (advanced)
A file called "ecbuild-cache.cmake" is generated during configuration. This file can be moved to a safe location, and specified for future builds to speed up checking of compiler/platform capabilities. Note that this is only accelerating fresh builds, as cmake internally caches also. Therefore this option is not recommended.

--build-cmake=[<prefix>]
Automatically download and build CMake version 3.5.2.
Requires an internet connection and may take a while. If no prefix is given, install into /Users/stephenh/projects/jedi-docs/docs.

--dryrun
Don't actually execute the cmake call, just print what would have been executed.

Available values for "cmake-argument":

Any value that can be usually passed to cmake to (re)configure the build.
Typically these values start with "-D".

example:  -DENABLE_TESTS=ON  -DENABLE_MPI=OFF  -DECKIT_PATH=...

They can be explicitly separated from [option...] with a "--", for the case there is a conflicting option with the "cmake" executable, and the latter’s option is requested.

NOTE: When reconfiguring a build, it is only necessary to change the relevant options, as everything stays cached. For example:

> ecbuild --prefix=PREFIX .
> ecbuild -DENABLE_TESTS=ON .

Compiling:

To compile the project with <N> threads:
> make -j<N>

To get verbose compilation/linking output:
> make VERBOSE=1

Testing:

To run the project's tests
> ctest

Also check the ctest manual/help for more options on running tests

Installing:

To install the project in location PREFIX with}
"--prefix=PREFIX" or
"-DCMAKE_INSTALL_PREFIX=PREFIX"
> make install

For examples on how to use ecbuild to compile JEDI bundles, see Building and Compiling JEDI (Step 3).

You can pass cmake command line options to cmake with ecbuild by proceeding them with two dashes --. For example, to use the cmake -Dtrace option mentioned above (useful for debugging), you can enter:

ecbuild -- --trace <path_to_bundle> # example that adds the --trace option to the cmake call

It is recommended to choose one of the JEDI repositories and look through all of the CMakeLists.txt files. This will help you get oriented in how these files are used to piece together the build, test and install flows. You will notice ecbuild macros (with names starting with "ecbuild_") along with native cmake commands.
1.3.7 Debugging Tools

Under construction

kdbg

Under construction

1.4 JEDI Environment

1.4.1 Singularity

In order to appreciate what Singularity is and why we use it, it is best to begin with its purpose. From a JEDI perspective, the purpose of Singularity is to provide a uniform computing environment (software tools, libraries, etc) that will enable users like you to easily build, compile, and run JEDI across a wide range of computing platforms, from laptops and workstations running Mac OS X or Linux to parallel High-Performance Computing (HPC) systems such as Theia (NOAA), Discover (NASA), or Cheyenne (NCAR).

The computing environment that Singularity creates is called a container. There are other container providers as well, the most well-known being Docker. An advantage of Singularity over these other alternatives is that it was specifically designed to address the challenges associated with HPC systems and collaborative software development efforts that require full control over your own software stack (see this article in HPCWire and this article in “Cloud Computing for Science and Engineering”).

In contrast to virtual machines, containers do not include the necessary software to build an entire operating system. Rather, they work with the host operating system to provide the desired functionality, including the libraries, applications, and other software tools that your code needs to run. So containers generally require much less memory to store and to set up than virtual machines. Singularity encapsulates your software environment in a single disk image file that can be copied to and invoked on any system on which Singularity itself is installed. The JEDI environment is contained in one such image file (see below).

Installing Singularity

If you are using a Vagrant virtual machine that you created with the JEDI Vagrantfile as described on our Vagrant page, then you can skip this step: Singularity 3.0 is already installed. Or, if you’re running JEDI somewhere other than your personal computer, Singularity may already be installed.

You can check to see if Singularity is already installed (and if it is, which version is installed) by typing

```
singularity --version
```

If it is not installed, and if you have root privileges, then you can install it yourself as described in the remainder of this section.

If Singularity is not installed and if you cannot install it because you do not have root privileges, then we recommend that you use Charliecloud instead. Root privileges are not needed to install and use the JEDI Charliecloud container and it provides the same software libraries as the Singularity container. Alternatively, you may ask your system administrators to install Singularity. However, many HPC centers will be reluctant to do so because of security vulnerabilities.

If you do wish to proceed with Singularity, we recommend that you install the most recent stable release (version 3.0 as of Dec, 2018).
If an up-to-date version of Singularity is already installed on your system, you can skip ahead to Building the JEDI Environment. The instructions that follow are mainly intended to help you install Singularity on your own workstation or laptop.

As noted above, Singularity is not a virtual machine so it does not build its own operating system. Instead, it must work with the host operating system. Like Charliecloud, Singularity relies on linux mount namespaces in order to set up application environments that are isolated from the host. Neither Mac OS X nor Windows currently supports mount namespaces.

So, if you are running Mac OS or Windows, then you must first set up a linux environment. This requires a proper virtual machine (VM). The recommended VM provider is Vagrant by HashiCorp, which can build and configure an appropriate linux operating system using Oracle’s VirtualBox software package.

In short, Vagrant and VirtualBox provide the linux operating system while Singularity (or Charliecloud) provides the necessary software infrastructure for running JEDI (compilers, cmake, ecbuild, etc) by means of the JEDI singularity image.

Singularity offers comprehensive installation instructions for Singularity 3.0 and we refer JEDI users to that site for the most up-to-date information and for troubleshooting. Here we summarize the main steps.

The first step is to make sure you have the correct dependencies. On ubuntu systems, you can install them by copying and pasting this:

```
# for ubuntu
sudo apt-get update
sudo apt-get install -y build-essential libssl-dev
sudo apt-get install -y uuid-dev libgpgme11-dev squashfs-tools
```

Next you need to install and configure the Go programming language, which Singularity 3.0 requires. There are multiple ways to do this but this should work on most linux systems (note - this installs in /usr/local, which requires root privileges):

```
export VERSION=1.11.2 OS=linux ARCH=amd64
wget https://dl.google.com/go/go$VERSION.$OS-$ARCH.tar.gz
sudo tar -C /usr/local -xzf go$VERSION.$OS-$ARCH.tar.gz
echo 'export GOPATH=${HOME}/go' >> ~/.bashrc
echo 'export PATH=/usr/local/go/bin:${PATH}:${GOPATH}/bin' >> ~/.bashrc
source ~/.bashrc
```

You can enter `go help` to see if this installation worked.

Now clone the Singularity repository from GitHub and `go get` its dependencies:

```
mkdir -p $GOPATH/src/github.com/sylabs
cd $GOPATH/src/github.com/sylabs
git clone https://github.com/sylabs/singularity.git
cd singularity
go get -u -v github.com/golang/dep/cmd/dep
```

Now you can compile and install Singularity (requires root privileges)

```
cd $GOPATH/src/github.com/sylabs/singularity
./mconfig
make -C builddir
sudo make -C builddir install
```
Building the JEDI environment

Once singularity is installed on your system, the rest is easy. The next step is to download the JEDI Singularity image from the singularity hub (shub):

```bash
singularity pull shub://JCSDA/singularity
872.87 MiB / 872.87 MiB
100.00% 17.98 MiB/s 48s
```

Strictly speaking, you only have to do this step once but in practice you will likely want to update your JEDI image occasionally as the software environment continues to evolve. The pull statement above should grab the most recent development version of the JEDI image file (it may take a few minutes to execute).

The name of the image file may vary depending on your version of Singularity and the name of the file on the Singularity Hub (shub). For example, if you are running Singularity version 2.4 or 2.6, the above command may retrieve a file called JCSDA-singularity-master-latest.simg. In Singularity version 3.0, it may be called singularity_latest.sif. In what follows, we will represent this name as `<image-file>` - you should replace this with the name of the file retrieved by the pull command.

Though you can execute individual commands or scripts within the singularity container defined by your image file (see the exec and run commands in the Singularity documentation), for most JEDI applications you will want to invoke a singularity shell, as follows:

```bash
singularity shell -e <image-file>
```

Now you are inside the Singularity Container and you have access to all the software infrastructure needed to build, compile, and run JEDI. The `-e` option helps prevent conflicts between the host environment and the container environment (e.g. conflicting library paths) by cleaning the environment before running the container. Note that this does not mean that the container is isolated from the host environment; you should still be able to access files and directories on your host computer (or on your virtual machine if you are using Vagrant) from within the Singularity container.

If you installed singularity from within a Vagrant virtual machine (Mac or Windows), then you probably set up a a `/home/vagrant/vagrant_data` directory (you may have given it a different name and/or path) that is shared between the host machine and the virtual machine. Since this is mounted in your home directory, you should be able to access it from within the container. However, sometimes you may wish to mount another directory in the container that is not accessible from Singularity by default. For example, let’s say that you are working on an HPC system and you have a designated workspace in a directory called `$SCRATCH`. We have included a mount point in the JEDI singularity container called `/worktmp` that will allow you to access such a directory. For this example, you would mount your work directory as follows:

```bash
singularity shell --bind $SCRATCH:/worktmp -e <image-file>
```

After you enter the container you can cd to `/worktmp` to access your workspace.

There is another “feature” of Singularity that is worth mentioning. Though Singularity starts a bash shell when entering the container, You may notice that it does not call the typical bash startup scripts like `.bashrc`, `.bash_profile` or `.bash_aliases`. Furthermore, this behavior persists even if you do not use the `-e` option to singularity shell. This is intentional. The creators of Singularity deliberately arranged it so that the singularity container does not call these startup scripts in order to avoid conflicts between the host environment and the container environment. It is possible to circumvent this behavior using the `--shell` option as follows:

```bash
# NOT RECOMMENDED!
singularity shell --shell /bin/bash -e <image-file>
```

However, if you do this, you may begin to appreciate why it is not recommended. In particular, you’ll notice that your command line prompt has not changed. So, it is not easy to tell whether you are working in the container or not. Needless to say, this can get very confusing if you have multiple windows open!
It is safer (and only minimally inconvenient) to put your aliases and environment variables in a shell script and then just get in the habit of sourcing that script after you enter the container, for example:

```
source startup.sh
```

where `startup.sh` contains, for example:

```
#!/bin/bash
alias Rm='rm -rf '
export FC=mpifort
export DISPLAY=localhost:0.0
```

The last two lines of this example script are particularly noteworthy. Setting the FC environment variable as shown is currently required to compile and run JEDI with multiple mpi threads. And, setting the DISPLAY environment variable as shown should enable X forwarding from the Singularity container to your computer if you are using linux/unix. This in turn will allow you to use graphical tools such as `emacs` or `kdbg`.

If you are invoking the singularity shell from a vagrant virtual machine, then X Forwarding is a bit more complicated; See here for how to setup X Forwarding on a Mac.

For a full list of options, type `singularity shell --help` from outside the container.

To exit the Singularity container at any time, simply type

```
exit
```

If you are using a Mac, you may wish to type `exit` a second time to exit Vagrant and then shut down the virtual machine with `vagrant halt` (See Working with Vagrant and Singularity).

### 1.4.2 Charliecloud

Charliecloud is similar to `Singularity` in that it is a software container service. However, it has one significant advantage over Singularity in that it does not require root privileges to install the Charliecloud software or to run Charliecloud containers. This can be a huge advantage, particularly on HPC systems where users generally do not have root privileges and where system administrators are often reluctant to install singularity because of security concerns.

Charliecloud was developed and is still maintained by software developers at Los Alamos National Laboratory. We refer you to to their Documentation pages for further information about the project, for instructions on how to build containers and other resources, and for troubleshooting any problems. Note in particular the Charliecloud command reference.

In the documentation that follows we focus only on what you need to know as a user of JEDI.

**Warning:** See the Charliecloud documentation (particularly the Installation and Tutorial sections) for tips and warnings about using Charliecloud. For example, there is a known bug on the Cray Linux Environment that causes nodes to crash just before exiting some Charliecloud jobs (they do have a workaround). Also, if you have a large number of MPI processes that each invokes the same Charliecloud container, it could overwhelm a network file system. For JEDI, we’ve run MPI jobs from within the container and have not yet noticed a problem with this (please let us know if you do!).

**Installing Charliecloud**

If you are using a Vagrant virtual machine that you created with the JEDI Vagrantfile as described on our Vagrant page, then you can skip this step: Charliecloud is already installed.
The Charliecloud Documentation pages have thorough Installation Instructions. This is the most up-to-date documentation available and if you have any problems with the procedure describe here we refer to you that page for troubleshooting.

### Installing on Mac OS and Windows systems

Like Singularity, Charliecloud relies on linux mount namespaces to control the set of filesystem mounts that are visible to a process. Neither Mac nor Windows operating systems currently support such mount namespaces.

So, if you are using a Mac or Windows laptop, we recommend that you install a virtual machine provider like Vagrant. Instructions on how to do so are provided on our JEDI Vagrant Page. This will allow you to create a linux virtual machine on your computer where you can install Charliecloud and run JEDI.

You are free to choose what type of linux operating system you want to have for your Vagrant virtual machine. In much of what follows, we will use ubuntu as an example, and in particular ubuntu 18.04. We recommend this environment for running Charliecloud, in part because it has a more up-to-date version of bash than the ubuntu 16.04 bento box available from virtualbox. However, most recent linux varieties (including ubuntu 16.04) should be sufficient, as long at they support mount namespaces (see the linux installation section for further information).

### Installing on Linux Systems

To install Charliecloud, you’ll need a relatively recent version of linux that is capable of creating mount namespaces. Most linux implementations from the last few years should be sufficient. If you want to check the version of the linux kernel, you can enter:

```
uname -r
```

Charliecloud recommends a version of 4.4 or later.

You’ll also need a C compiler and some basic software tools. These may already be installed if you are using an HPC system or a linux PC. However, if you are starting from a bare installation, such as a Vagrant virtual machine or a cloud computing instance (e.g. Amazon EC2), you may need to install these yourself. On ubuntu or Debian systems you can do this with:

```
sudo apt-get update
sudo apt-get install build-essential python
```

Or, on CentOS, Fedora, Amazon Linux, and similar systems, you might instead enter:

```
sudo yum update
sudo yum groupinstall "Development Tools"
```

These commands require root access. If you do not have root access, chances are good that the required software is already installed.

The next step is to clone the Charliecloud repository on GitHub, build it, and install it into a directory of your choice. Here we build and install the code into the user’s home directory:

```
mkdir ~/build
cd ~/build
git clone --recursive https://github.com/hpc/charliecloud.git
cd charliecloud
make
make install PREFIX=$HOME/charliecloud
```
Unless there were problems, Charliecloud should now be installed in the user’s home directory, in the subdirectory charliecloud. If you wish to test the installation (optional), run the Bats test suite as described in the Charliecloud Documentation.

Now add the Charliecloud executables to your path. You may wish to do this interactively when you install Charliecloud for the first time but we recommend that you also put it in a startup script such as .bash_profile.

```
export PATH=$PATH:$HOME/charliecloud/bin
```

**Note:** If you do decide to run the Charliecloud test suite you should be aware that some of these tests require root privileges. If you do not have root privileges, you can disable these tests by setting this environment variable before running make test:

```
export CH_TEST_PERMDIRS=skip
```

### Building the JEDI environment

Once Charliecloud is installed on your system, the next step is to make a home for the JEDI Charliecloud container and download it as follows (you may also have to install wget if it’s not included in the developer tools mentioned above):

```
mkdir -p ~/jedi/ch-container
cd ~/jedi/ch-container
wget http://data.jcsda.org/containers/ch-jedi-latest.tar.gz
```

This looks like a normal gzipped tar file. However, **you should not unpack it with** `tar`! Instead, unpack it with this command:

```
ch-tar2dir ch-jedi-latest.tar.gz .
```

This may take a few minutes so be patient. When done, it should give you a message like ./ch-jedi-latest unpacked ok and it should have created a directory by that same name. In our example, this directory would be located in ~/jedi/ch-container/ch-jedi-latest.

This is the JEDI Charliecloud container. It’s functionally equivalent to a Singularity image file but it appears as a directory rather than a single file. Furthermore, that directory contains a complete, self-contained linux filesystem, complete with its own system directories like /usr/local, /bin, and /home.

To enter the Charliecloud container, type:

```
ch-run -c $HOME ~/jedi/ch-container/ch-jedi-latest -- bash
```

Let’s reconstruct this command to help you understand it and customize it as you wish.

The `ch-run` command runs a command in the Charliecloud container.

The `-c $HOME` option tells Charliecloud to enter the container in the user’s home directory, which is the same inside and outside the container. If this option is omitted, you will enter the container in the root directory. Typing `cd` will then place you in your home directory.

The `~/jedi/ch-container/ch-jedi-latest` argument is the name of the container you want Charliecloud to run. This is the name of the directory created by the `ch-tar2dir` command above. If you run this from the container’s parent directory, in this case ~/jedi/ch-container, then you can omit the path.

Finally, we have to tell `ch-run` what command we want it to run. The command (including options and arguments) that comes after the double hyphen -- will be executed within the container. If you were to run a single command,
like `ls -alh`, then `ch-run` will enter the container, execute the command, and exit. However, in this example, we started up a bash shell, with `-- bash`. So, **all commands that follow will be executed inside the container. In order to exit the container, you have to explicitly type exit.** This brings us to this important warning:

**Warning:** When you enter the Charliecloud container, your prompt may not change!! So, it can be very difficult to tell whether or not you are in the Charliecloud container or not. One trick is to enter the command `eckit-version`. If you do not have eckit installed on the host system (which may be a vagrant virtual machine or an amazon EC2 instance), then this command will only return a valid result if you are indeed inside the Charliecloud container. Note that this is different from Singularity, which does change your prompt when you enter the container.

Now, since you are in the container, you have access to all the software libraries that support JEDI. You can now proceed to build and run JEDI as described elsewhere in this documentation.

For example, to run and test ufo-bundle, you can proceed as follows:

```bash
git config --global credential.helper 'cache --timeout=3600'
mkdir -p ~/jedi/src
cd ~/jedi/src
git clone https://github.com/JCSDA/ufo-bundle.git
mkdir -p ~/jedi/build
cd ~/jedi/build
export FC=mpifort
ecbuild ../src/ufo-bundle
make -j4
ctest
```

**Warning:** On some systems (notably Cheyenne) it may be necessary to explicitly add `/usr/local/lib` to your `LD_LIBRARY_PATH` environment variable within the Charliecloud container, as follows:

```bash
export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/usr/local/lib
```

**General Charliecloud Tips**

If you’re running a Charliecloud container from within Vagrant, the most important tip when using Charliecloud (because it is easy to forget) is to **remember to type exit twice** when you are finished working; once to leave the Charliecloud container and a second time to leave Vagrant.

Another important thing to realize (whether you are running Charliecloud from Vagrant, from AWS, from an HPC system, or from anywhere else), is that many directories on the host are still visible to you from within the container. This includes your home directory. So, it is easy to access files from within the container - you should be able to see and edit everything in your home directory.

In addition to the user’s home directory, a few system directories are also mounted and accessible from within the container. This includes `/dev`, `/proc`, and `/sys`. But, notably, it does not include `/usr/local`; This is the whole point of the container - to re-define the software that is installed on your system without conflicting with what you have installed already.

These mounted directories should be sufficient for many users. However, you have the option to also mount any additional directories of your choice. An important example is for Mac or Windows users who run Charliecloud from within a Vagrant virtual machine. The Vagrant home directory is visible from within the Charliecloud container but this directory is typically not accessible from the host operating system, e.g. MacOS.

1.4. JEDI Environment
In our Vagrant documentation we describe how you can set up a directory that is shared between the host system (Mac OS) and the virtual machine (Vagrant). From within Vagrant, we called this directory /home/vagrant/vagrant_data. Since this is in our home directory, it should be visible already from within the Charliecloud container so no explicit binding is necessary.

However, what if we were to instead mount the shared directory in /vagrant_data (as viewed from Vagrant)? This is the default behavior in the Vagrantfile as created by the `vagrant init` command. Since this branches off of the root directory, it would not be visible by default from within the Charliecloud container. However, you can still mount this (or nearly any other directory of your choice) in the Charliecloud container using the `-b` (or `--bind`) option:

```
ch-run -b /vagrant_data -c $HOME ch-jedi-latest -- bash
```

By default, this is mounted in the Charliecloud container as the directory /mnt/0. You can change the mount point provided that the target directory already exists within the container.

For example, if you create a directory called /home/vagrant/vagrant_data before entering the container, then you can identify that directory as the target for the mount:

```
ch-run -b /vagrant_data:/home/vagrant/vagrant_data ch-jedi-latest -- bash
```

Then, when you are inside the container, any files that you put in /home/vagrant/vagrant_data will be accessible from Mac OS.

**Tips for HPC Systems**

By default, Charliecloud does not change environment variables (with a few exceptions). The JEDI Charliecloud container does explicitly set a few variables such as `NETCDF`, `FC`, `PIO`, etc. (for bash shells) but it’s still good practice to clean your environment by purging other modules before you enter your `ch-run` command. Most HPC systems use some form of environment modules to load software packages. So “cleaning your environment” usually just looks like this:

```
module purge
```

Another common practice on HPC systems is to run applications in designed work or scratch directories instead of one’s home directory. This is often required to have access to sufficient disk space. The JEDI Charliecloud and Singularity containers includes a `/worktmp` directory that can be used as a mount point for a system work space. For example, on Cheyenne one may wish to do this:

```
ch-run -b/glade/work/`whoami`:/worktmp <path>/ch-jedi-latest -- bash
```

This is good, but for substantial parallel applications there is an approach that is even better for MPI jobs. System administrators at HPC centers spend a lot of time and effort configuring their MPI implementations to take full advantage of the system hardware. If you run the mpi implementation inside the container (currently openmpi), you won’t be able to take advantage of these site-specific configurations and optimizations. Fortunately, there is a way out of this dilemma: you can invoke the parallel process manager, `mpirun` or `mpiexec` outside the container and then have each MPI process enter its own container. Again using Cheyenne as an example, you can do this in a batch script like this:

```
#!/bin/bash
#PBS -N multicon
#PBS -A <account-number>
#PBS -l walltime=00:10:00
#PBS -l select=4:ncpus=36:mpiprocs=36
#PBS -q regular
#PBS -j oe
```

(continues on next page)
#PBS -m abe
#PBS -M <email-address>

module purge
module load gnu/7.3.0 openmpi/3.1.3

export CHDIR=$HOME/ch-jedi
export WORK=/glade/work/`whoami`
export RUNDIR=/worktmp/jedi/fv3-gnu-openmpi/build/bin

### Run the executable

mpirun -np 144 ch-run -b $WORK:/worktmp -c $RUNDIR $CHDIR/ch-jedi-latest -- $BINDIR/

→ fv3jedi_var.x -- testinput/3dvar_c48.yaml

There are a few things to note about this example. First, mpirun is called from outside the container to start up 144 mpi tasks. Each task then starts its own Charliecloud container by running ch-run, mounting a work disk that is accessed through /worktmp in the container, as described above. The -c $RUNDIR option tells Charliecloud to cd to the $RUNDIR directory to run the command (note that this is the path as viewed from within the container). As before, the command appears after the --. But instead of entering the container by invoking a bash script, we enter a single command, which is here enclosed by double quotes ". So, in short, we are telling each MPI task to run this command in the container, from the $RUNDIR directory.

**Important** This will only work if the MPI implementations inside and outside the container are compatible. Since the MPI implementation inside the container is openmpi compiled with gnu compilers, we load the gnu/7.3.0 and openmpi/3.1.3 modules before calling mpirun.

This is usually more efficient than the alternative of running a single container with multiple mpi jobs:

```bash
export TMPDIR=/worktmp/scratch
ch-run -b $WORK:/worktmp -c $WORKDIR $CHDIR/ch-jedi-latest -- mpirun -np 144 $BINDIR/

→ fv3jedi_var.x -- testinput/3dvar_c48.yaml
```

This example illustrates another important tip to keep in mind. Openmpi uses the directory $TMPDIR to store temporary files during runtime. On Cheyenne, this is set to /glade/scratch/`whoami` by default. But this directory is not accessible from the container so, unless we do something about this, our executable will fail. Redefining it as /worktmp/scratch as shown here does the trick, provided that associated external directory $WORK/scratch exists. Recall that Charliecloud does not change environment variables so we can set it outside the container as shown. A similar workaround may also be required on other HPC systems.

### 1.4.3 Vagrant (for Mac and Windows)

In order to set up the JEDI environment with Singularity or Charliecloud, you’ll need to first set up a linux operating system. We will often focus on ubuntu for illustration purposes but if you prefer other varieties of linux, these are also available from the Virtualbox provider that we describe below.

So, if you’re using a Mac or Windows computer, you will want to set up a local, self-contained linux environment within your broader operating system. This is commonly called a virtual machine (VM). Once you have a linux VM up and running, you will be able to install Singularity or Charliecloud and triumphantly enter the corresponding JEDI Container.

This can all be achieved using an application called Vagrant, which is developed and distributed by a company called Hashicorp. Though we will sometimes refer to Vagrant as the virtual machine provider, the actual linux operating system is ultimately provided by Oracle’s VirtualBox software package. Vagrant is essentially a tool will allow you to build, configure, and manage a VirtualBox operating system. In particular, we will use Vagrant and VirtualBox to
create an Ubuntu virtual machine on your Mac workstation or laptop. We’ll then populate that Linux VM with specific JEDI software tools (compilers, ecbuild, etc.) using Singularity or Charliecloud.

From this brief introduction, it is clear that you do not need to worry about Vagrant or VirtualBox if your working machine (whether it is a workstation, laptop, or HPC system) is already running Linux and/or is already running Singularity. By *working machine* we mean whatever machine you plan to compile and run JEDI on.

You *do* need Vagrant and VirtualBox (or something equivalent) if you wish to run JEDI from a Mac or Windows machine. Though you can use Vagrant for both platforms, we focus here on Mac OS X.

We refer Windows users to the Vagrant download page where you can download a binary implementation for Windows and install it using the Windows package manager. After installing Vagrant, you may wish to return to this document for tips on how to configure it for JEDI (skipping Step A of the next section).

**Installing and Configuring Vagrant**

**A: Install Vagrant and VirtualBox**

As with Windows, you can install Vagrant on a Mac by downloading a pre-compiled binary package from the Vagrant Download Page. However, we recommend that you install with Homebrew as described below to give you more flexibility in managing both vagrant and virtualbox.

Before you begin you should install or update Homebrew. You’ll need a relatively recent version in order to use the cask extension. Once you have done this, you can proceed as follows:

```
brew cask install virtualbox
brew cask install vagrant
brew cask install vagrant-manager
```

**B: Download JEDI Configuration File**

Now we need to tell Vagrant what type of virtual machine we want to create and how to provision it with the software we need. This is done by means of a configuration file that goes by the default name of Vagrantfile.

So, to proceed, you should first create a directory where you will place your Vagrantfile. This is where you will launch your virtual machine. **You should also create a subdirectory called vagrant_data that we will use.** If you don’t create this directory, you will get an error when vagrant tried to mount it.

You can call the parent directory whatever you wish but if you change the name of the vagrant_data directory then you will also have to change the Vagrantfile.

```
mkdir $HOME/jedi-vm
cd $HOME/jedi-vm
mkdir vagrant_data
```

In what follows, we will refer to this as the home directory of your Vagrant Virtual Machine (VM).

We at JCSDA provide a Vagrantfile that can be used to create a virtual machine that is pre-configured to build and run JEDI, with both Singularity and Charliecloud pre-installed.

Download the JEDI Vagrantfile here

Or, alternatively, you can retrieve it with

```
wget http://data.jcsda.org/containers/Vagrantfile
```
Place this Vagrantfile in the home directory of your Vagrant VM.

**Warning:** If you already have a Vagrant VM installed and you want to install a new one (particularly using a Vagrantfile in the same directory as before), then you may have to fully delete the previous VM first to avoid any conflicts. Instructions on how to do this are provided in the *Deleting a Vagrant VM* section below.

**Note:** If you have problems with this JEDI Vagrantfile, there is an alternative Vagrantfile that you can download that expands the disk storage using the `disksize` plugin to Vagrant. This also comes with Charliecloud and Singularity pre-installed. After downloading this file, it’s easiest to change its name to `Vagrantfile` and then run `vagrant up` again. However, before trying this make sure that you either destroy your previous VM or create the new VM from a different directory and give it a different name (edit the Vagrantfile and search for `jedibox`).

### C: Launch your Virtual Machine

Now you are ready to create your virtual machine by running this command:

```
vagrant up
```

The first time you run this command, it will take several minutes. Vagrant is installing Singularity, Charliecloud, and a few other supporting software packages. Once created, these will be part of your virtual machine and they do not need to be re-installed (unless you explicitly tell vagrant to do so).

So, when this command finishes, you can log into your virtual machine with

```
vagrant ssh
```

Now you are in a linux environment (CentoOS 7). From here you can pull the JEDI container of your choice,

- Click here to proceed with JEDI Singularity Container
- Click here to proceed with the JEDI Charliecloud Container

The choice is up to you. Both the Singularity container and the Charliecloud container are built from the same Docker image file so they contain identical software. The main advantage to using Charliecloud is that you do not need root privileges to run it. But, if you use Vagrant this should not be a problem because you should have root privileges in your Vagrant VM. You can even try both in the same virtual machine and see which one you prefer.

Depending on which Vagrantfile you use, your VM may run either the ubuntu or the CentOS operating system. However, you shouldn’t need to be too concerned about this because you’ll be working mostly in either the Singularity container or the Charliecloud container which both run ubuntu. So, if you work within the container, you will be in an ubuntu environment regardless of which OS your vagrant VM is running.

### D: Exit Container and Vagrant

Normally you will be spending your time working in either the Singularity container or the Charliecloud container. When you’re finished working for the day, it’s important to remember to enter `exit` twice, once to exit the container and once to log out of the Vagrant virtual machine:

```
exit # to exit Singularity or Charliecloud
exit # to exit Vagrant
```

Now, to temporarily shut down your virtual machine, enter
vagrant halt

Note that this is very different than the vagrant destroy command, which should be used with caution. As the name of the command suggests, vagrant destroy will completely destroy the virtual machine along with all the files and data it contains. So, if you do this, you will have to re-create the virtual machine and re-install any JEDI bundles that you are working with. And, you will lose any files that you have been editing. By contrast, vagrant halt will merely shut down the virtual machine, retaining all your files. This will allow you to gracefully log out of your workstation or laptop without harming your JEDI environment. For further details see the Vagrant command reference.

E: Enable X Forwarding (Optional)

If you’d like to use graphical tools such as kdbg or emacs from within the Singularity or Charliecloud container, you will need to set up X forwarding. If you’re doing this on a Mac, you will first need to install XQuartz, if it’s not already installed.

After XQuartz is up and running, you can create and enter your VM as described in step C above. Next you will have to set your DISPLAY environment variable to use your local machine. This is best done from within the container (either Singularity or Charliecloud) because environment variables set outside the container may not be accessible from within.

```bash
# inside the container
export DISPLAY=10.0.2.2:0.0
```

You may wish to add the appropriate display definition to an initialization script that you can run every time you enter the singularity container as described here. Then, enter this on your host machine (i.e. your Mac or Windows machine), to grant the VM permission to display

```bash
#On your Mac
xhost + 127.0.0.1
```

These are the addresses that Vagrant uses for by default. You may wish to add the appropriate display definition to an initialization script that you can run every time you enter the singularity container as described here.

To test the display, you can start a graphical application. For example:

```bash
# inside the container
emacs &
```

Troubleshooting Tips

If the above procedure did not work, there are several things to try.

First, if you have a Mac, make sure XQuartz is installed. You may need to re-boot your VM for a new installation to take effect.

Next, try running emacs from outside the container to see if the problem is with Vagrant or with the container.

If you used a different Vagrant box than the one specified in the JEDI Vagrantfile (for example, if you used one from Singularityware), if might help to set your DISPLAY variable in the container to this instead:

```bash
export DISPLAY=localhost:10.0
```

If the display still does not work, then you may need to explicitly grant Vagrant access to your display through xauth as we now describe.

Exit the container and exit vagrant. Then edit your Vagrantfile and add these two lines (at the bottom, just before the end in the main Vagrant.configure("2") do |config| loop will do)
config.ssh.forward_agent = true
config.ssh.forward_x11 = true

Then recreate your vagrant VM, log in, and enter the container (for example, for Singularity):

```bash
vagrant halt  # restart vagrant
vagrant up
vagrant ssh
singularity shell --bind ./vagrant_data -e <singularity-image-file>
```

Now create an .Xauthority file and generate an authorization key for your display:

```bash
touch ~/.Xauthority
xauth generate 10.0.2.2:0.0 . trusted
```

You can list your new authorization key as follows:

```bash
xauth list
```

There should be at least one entry, corresponding to the display you entered in the `xauth generate` command above (you can ignore other entries, if present). For example, it should look something like this:

```
10.0.2.2:0 MIT-MAGIC-COOKIE-1 <hex-key>
```

where `<hex-key>` is a hexadecimal key with about 30-40 digits. Now, copy this information and paste it onto the end of the `xauth add` command as follows:

```bash
xauth add 10.0.2.2:0 MIT-MAGIC-COOKIE-1 <hex-key>
```

If all worked as planned, this should grant permission for vagrant to use your display.

**Customizing the Vagrantfile (optional)**

The JEDI Vagrantfile you downloaded in Step B above is already provisioned with everything you need to run JEDI, by means of the Singularity or Charliecloud software containers.

However, it’s useful to point out a few configuration options that some users may wish to customize.

**Creating your own Vagrantfile**

First comes the choice of machine. The JEDI Vagrantfile uses a CentOS 7 operating system but there are a number of other options available, particularly with the well-maintained bento boxes provided by Vagrant. You may wish to maintain multiple virtual machines with different linux operating systems.

For example, you can create your own Vagrantfile by entering something like this:

```bash
vagrant init bento/ubuntu-18.04
```

When you then run `vagrant up`, this will create an ubuntu 18.04 operating system. You can then install either Singularity or Charliecloud manually.

The makers of Singularity also provide their own Vagrant box, with Singularity pre-installed:

```bash
vagrant init singularityware/singularity-2.4
```
However, as of Dec, 2018, the most recent version of Singularity available is 2.4; there have been considerable changes since then with the release of Singularity 3.0. Using the JEDI Vagrantfile will ensure that your version of Singularity is compatible with the version used to create the JEDI Singularity image.

**Allocating Resources for your Virtual Machine**

The JEDI Vagrantfile comes pre-configured to allocate 4GB of memory and 6 virtual CPUS to the VM. This is the minimum resource allocation to run ufo-bundle. Other bundles such as fv3 may require more memory (as much as 12-16 GB) and/or more vCPUs. Furthermore, if you create your own Vagrantfile, the default resource allocation will likely be insufficient to run JEDI.

You can change these resource allocations by editing the Vagrantfile. Look for the following section that specifies the provider-specific configuration (variable names may differ). Change the `vb.memory` (in MB) and `vb.cpus` fields as shown here:

```ruby
config.vm.provider "virtualbox" do |vb|
  # [...]
  # Customize the amount of memory on the VM:
  vb.memory = "4096"

  # Customize the number of cores in the VM:
  vb.cpus = "6"

  # [...]
end
```

**File transfer between your Mac and the VM**

In Step B above we created a directory called `vagrant_data`. The JEDI Vagrantfile is configured to use this directory to transfer files between your host machine (which may be running Mac OS or Windows) and your VM. Within the VM, this directory is mounted as `$HOME/vagrant_data`.

To change this, you can edit the Vagrantfile and find the section for a `synced folder`:

```ruby
# Share an additional folder to the guest VM. The first argument is
# the path on the host to the actual folder. The second argument is
# the path on the guest to mount the folder. And the optional third
# argument is a set of non-required options.

c.vm.synced_folder "vagrant_data", "/home/vagrant/vagrant_data"
```

The first argument specifies the directory on the host machine, relative to the home directory of your Vagrant VM (i.e. the directory where the Vagrantfile is). The second specifies the path of the directory on the VM. You can change these paths and/or names if you wish but **make sure the host directory exists before running vagrant up** so it can be properly mounted.

It might also be necessary to create the mount point from within the vagrant VM:

```bash
mkdir -/vagrant_data # from within the VM, if necessary
```

And, here is another tip: **Use an absolute path for your guest directory.** Vagrant will complain if you use a relative path, such as `.:/vagrant_data`. You will need root permission if you want to branch off of root (for example `/vagrant_data` is the default mounting if you run `vagrant init`.)
On a related note: your default user name when you enter Vagrant will be `vagrant` and your home directory will be `/home/vagrant`. If you want to change this you can do so by adding a line like this to your Vagrantfile:

```
config.ssh.username = 'vagabond'
```

For more information, and more options, see the Vagrant documentation.

**Working with Vagrant and the JEDI Container**

Once you have Vagrant and a container provider (either Singularity or Charliecloud) all set up as discussed above, your daily workflow may be as follows. You might start by going to the directory where you put your Vagrantfile. Then fire up and log in to your virtual machine.

```
cd $HOME/jedi-vm
vagrant up
vagrant ssh
```

From there you can enter the container and (optionally) run your startup script. For example, in the case of Singularity this would look something like this:

```
singularity shell -e <singularity-image-file>
source startup.sh
```

The equivalent commands for Charliecloud would be:

```
ch-run -c /home/vagrant ch-jedi-latest -- bash
source startup.sh
```

Now you’re in the JEDI container and you can do whatever you wish: edit files, build, compile and run JEDI, etc. If you want to use X-forwarding you’ll have to explicitly tell your Mac to accept graphical input from the Vagrant VM as described in *Step G* above:

```
#On your Mac
xhost + 127.0.0.1
```

You may be tempted to automate this so you don’t have to enter this command every time you start up your virtual machine. However, this is more subtle than you might expect. Since this is the IP address of localhoat, placing this command in your `.bash_profile` file might cause your terminal application to hang when you first start it up because localhost is not yet defined. You can avoid this by adding `xhost +` to your `.bash_profile` but be careful with this because it may open you up to security vulnerabilities by allowing clients to connect to your machine from any remote host. Entering the explicit command above or putting it in a bash script that you execute manually every time you log in is somewhat inconvenient but much safer.

When you’re done for the day you can exit and shut down the VM:

```
exit # to exit Singularity or Charliecloud
exit # to exit Vagrant
vagrant halt # to shut down the virtual machine
```

**Deleting a Vagrant VM**

When you shut down a Vagrant virtual machine (VM) with `vagrant halt`, it’s like shutting down your laptop or workstation. When you restart the VM, you can pick up where you left off. You’ll see all the files and directories that were there before.
This is usually desirable. However, it does mean that the VM is occupying disk space on your machine even when it is suspended. If you have created multiple VMs, this can add up. So, it is often useful to delete a VM if you are done using it.

To check vagrant’s status at any time enter:

```
vagrant global-status
```

This is a useful command to know about. It will tell you all the VMs vagrant knows about on your computer including the path where the Vagrantfile is located and the state. A `vagrant up` command will put the VM in a running state while a `vagrant halt` command will put the VM in a poweroff state.

If you want to delete one or more of these VMs, the first step is to save any files you have on the VM that you want to preserve. This can be done by moving them to the `~/vagrant_data` directory which will still exists on your local computer after the VM is deleted.

Now, the best way to proceed is to go to the directory where the vagrant file is and enter:

```
vagrant destroy # enter y at the prompt
rm -rf .vagrant
```

The first command deletes all of the disks used by the virtual machine, with the exception of the cross-mounted `vagrant_data` directory which still exists on your local computer. The second command resets the vagrant configuration. This is particularly important if you re-install a new VM where another VM had been previously. If you skip this step, `vagrant up` may give you errors that complain about mounting the `vagrant_data` directory (“...it seems that you don’t have the privileges to change the firewall…”).

This is a start, but you’re not done. As mentioned at the top of this document, Vagrant is really just an interface to VirtualBox, which provides the linux OS. The Virtualbox VM that contains the linux OS still exists and is still using resources on your computer. To see the VirtualBoxes that are currently installed though Vagrant, run:

```
vagrant box list
```

If you used the JEDI Vagrantfile as described in Step B above, then you’ll see one or more entries with the name `centos/7`. The first step here is to prune any that are not being used any more with:

```
vagrant box prune
```

However, even this might not delete the VM you want to delete. Run `vagrant list` to see if it is still there and if it is, you can delete it with:

```
vagrant box remove centos/7
```

..or ubuntu or singularityware or whatever name is listed for the box you want to delete.

In some cases it might also help to delete the hidden `.vagrant` file that is created by vagrant in the same directory as your Vagrantfile. So, from that directory, enter:

```
rm -rf .vagrant
```

Now, this should be sufficient for most situations. Most users can stop here with confidence that they have deleted their unwanted VMs and have freed up the resources on their local computer.

However, it is possible that there might still be VirtualBox VMs on your machine that Vagrant has lost track of. You might notice this if you try to create a new VM with `vagrant up` and it complains that “A VirtualBox machine with the name jedibox already exists” (or a similar error message).

If this is the case, you can run VirtualBox directly to manage your VMs. This can be done through the command line with the `vboxmanage` command (run `vboxmanage --help` for information) but we recommend the VirtualBox GUI, which is more user-friendly.
To access the GUI on a Mac or Windows machine, just go to your Applications folder and double click on the VirtualBox icon. There you will see a complete list of all the VirtualBox VMs installed on your system and you can delete any that you don’t want by selecting the **machine** menu item and then **remove**.

### 1.4.4 JEDI Modules

If you are running JEDI on a personal computer (Mac, Windows, orLinux) we recommend that you use either the **JEDI Singularity container** or the **JEDI Charliecloud container** `<charliecloud>`. These provide all of the necessary software libraries for you to build and run JEDI.

If you are running JEDI on an HPC system, **Charliecloud** is still a viable option. However, on selected HPC systems that are accessed by multiple JEDI users we offer another option, namely **JEDI Modules**.

Environment modules are implemented on most HPC systems and are an easy and effective way to manage software libraries. Most implementations share similar commands, such as:

```
module list  # list modules you currently have loaded
dmodule spider <string>  # list all modules that contain <string>
dmodule avail  # list modules that are compatible with the modules you already have
module load <package1> <package2> <...>  # load specified packages
dmodule unload <package1> <package2> <...>  # unload specified packages
dmodule swap <packageA> <packageB>  # swap one module for another
dmodule purge  # unload all modules
```

For further information (and more commands) you can refer to a specific implementation such as **Lmod**.

We currently offer JEDI modules on several HPC systems, as described below. Consult the appropriate section for instructions on how to access the JEDI modules on each system.

These modules are functionally equivalent to the JEDI Singularity and Charliecloud containers in the sense that they provide all of the software libraries necessary to build and run JEDI. But there is no need to install a container provider or to enter a different mount namespace. After loading the appropriate JEDI module or modules (some bundles may require loading more than one), users can proceed to **compile and run the JEDI bundle of their choice**.

We begin with some general instructions on how to use modules that apply across all systems. We then give more detailed usage tips for specific systems.

**General Usage**

As a first step, it is a good idea to remove pre-existing modules that might conflict with the software libraries contained in the JEDI modules. Experienced users may wish to do this selectively to avoid disabling commonly used packages that are unrelated to JEDI. However, a quick and robust way to ensure that there are no conflicts is to unload all modules with

```
dmodule purge
```

The next step is then to access the JEDI modules with:

```
dmodule use -a <module-path>
```

The `<module-path>` where JEDI modules are installed is system-specific as described below.

All implementations will include a default jedi module that you can load with:

```
dmodule load jedi
```

---

**1.4. JEDI Environment 49**
This should be sufficient for most users. But, some users and developers may wish to use different libraries. For example, the default module on many systems uses the Intel compiler suite but you can switch to the GNU compiler suite by entering something like this:

```
module switch jedi jedi/gnu
```

This is equivalent to entering

```
module unload jedi
module load jedi/gnu
```

Of course, this example will only work if a module named `jedi/gnu` exists. There may be name variations across platforms depending on what software has been installed. To see what options are available for JEDI, enter

```
module spider jedi
```

This will include alternative versions of the main jedi module (indicated with a slash as in `jedi/gnu` above) and it may also include supplementary modules for specific bundles. These are usually indicated with a hyphen. So, in summary, the full procedure for initializing the environment for some arbitrary bundle `<A>` might look like this:

```
module purge
module use -a <module-path>
module load jedi
module load jedi-<A> # unnecessary for ufo-bundle and most others
```

The `jedi` module is really multiple nested modules. To list the modules you currently have loaded, enter

```
module list
```

When you are happy with this, you are ready to **build and run your JEDI bundle.**

**Theia**

Theia is an HPC system located in NOAA’s NESCC facility in Fairmont, WV. On Theia, users can access the installed `jedi` modules by first entering

```
module use -a /contrib/da/modulefiles
```

Current options for setting up the JEDI environment include (choose only one)

```
module load jedi # intel compiler suite
module load jedi/jedi-gcc-7.3.0 # GNU 7.3.0 compiler suite
module load jedi/jedi-gcc-8.2.0 # GNU 8.2.0 compiler suite
```

Some system-specific tips for Theia include:

- If you are using intel compilers, run `ecbuild` with the following option in order to make sure you have the correct run command for parallel jobs:

```
ecbuild -DMPIEXEC=$MPIEXEC ...
```

- Use up to 12 MPI tasks to speed up the compilation

```
made -j12
```
Cheyenne

Cheyenne is a 5.34-petaflops, high-performance computer built for NCAR by SGI. On Cheyenne, users can access the installed Jedi modules by first entering:

```
module purge
module use /glade/work/miesch/modules/modulefiles/core
```

Current options for setting up the JEDI environment include (choose only one):

```
module load jedi/gnu-openmpi  # GNU compiler suite and openmpi
module load jedi/gnu-mpt     # GNU compiler suite and mpt
module load jedi/intel-openmpi  # Intel compiler suite and openmpi
module load jedi/intel-mpt    # Intel compiler suite and mpt
```

- Run ecbuild with the following option

```
mkdir build; cd build
ecbuild <path_of_the_jedi_code>
```

- Use multiple threads to speed up the compilation

```
make -j4
```

**Warning:** Please do not use too many threads to speed up the compilation, Cheyenne system administrator might terminate your login node.

Discover

Discover is 90,000 core supercomputing cluster capable of delivering 3.5 petaflops of high-performance computing for Earth system applications from weather to seasonal to climate predictions. On Discover, users can access the installed JEDI modules by first entering:

```
module use -a /discover/nobackup/projects/gmao/obsdev/rmahajan/opt/modulefiles
```

Current options for setting up the JEDI environment include (choose only one):

```
module load apps/jedi/gcc-7.3    # GNU v7.3.0 compiler suite
module load apps/jedi/intel-17.0.7.259  # Intel v17.0.7.259 compiler suite
```

- Run ecbuild with the following option to provide the correct path for MPIEXEC

```
ecbuild -DMPIEXEC=$MPIEXEC <path_of_the_jedi_source_code>
```

- Use up to 12 MPI tasks to speed up the compilation

```
make -j12
```
1.5 Building, Testing, and Running JEDI

1.5.1 Building and compiling JEDI

As described in detail elsewhere, the procedure for building and compiling JEDI rests heavily on the software tools CMake and ecbuild, which make your life much easier. A typical workflow proceeds in the following steps:

1. Clone the desired JEDI bundle
2. Optionally edit the CMakeLists.txt file in the bundle to choose the code branches you want to work with
3. cd to the build directory and run ecbuild to generate the Makefiles and other infrastructure
4. Run make to compile the code
5. Run ctest to verify that the bundle is working correctly

In terms of the actual commands you would enter, these steps will look something like this:

```
cd <src-directory>
git clone https://github.com/JCSDA/ufo-bundle.git
cd <build-directory>
ecbuild <src-directory>/ufo-bundle
make -j4
cctest
```

In this document we describe Steps 1 through 4, including the various options you have available to you at each step. For a description of Step 5, see our page on JEDI unit testing.

You will probably only need to do Step 1 once. However, if you are a developer who is making changes to one or more JEDI repositories, you will likely find it useful to execute Steps 2 through 5 multiple times, with progressively increasing frequency. For example, if you are working with a single repository, you may only need to do Step 2 once in order to tell ecbuild to compile your local branch. And, you’ll only need to run ecbuild (Step 3) occasionally, when you make changes that affect the directory tree or compilation (for example, adding a file that was not there previously or substantially altering the dependencies). By comparison, you will likely execute Steps 4 and 5 frequently as you proceed to make changes and test them.

Precursor: System Configuration

Before jumping into the actual building of JEDI, we highly recommend that you read this section. This information will let you avoid the need to enter your GitHub password many times during the JEDI build process, which can be annoying to say the least. And, it will allow you to avoid errors when using a bundle that requires multiple MPI threads.

All JEDI repositories are stored and distributed by means of GitHub. If you have used git before, then you probably already have a .gitconfig configuration file in your home directory. If you have not already done so at some point in the past, you can create a git configuration file by specifying your GitHub username and email as follows:

```
git config --global user.name <username-for-github>
git config --global user.email <email-used-for-github>
```

This is a recommended action for any user of GitHub since it governs how you access GitHub with git. However, there is another action that you may not have set up previously but that will be immensely useful to all JEDI users and developers: tell GitHub to cache your GitHub credentials:

```
git config --global credential.helper 'cache --timeout=3600'
```
This tells GitHub to keep your GitHub login information for an hour, i.e. 3600 seconds (feel free to increase this time if you wish). If you don’t do this, you may regret it - you’ll have to enter your GitHub password repeatedly throughout the build process as ecbuild proceeds to clone multiple GitHub repositories.

The statement above should be sufficient on most systems. However, on some systems (particularly HPC systems with stringent security protocols), it may be necessary to explicitly give git permission to store your GitHub password unencrypted on disk as follows:

```
git config --global --add credential.helper 'store'
```

As for all your files, your password will still be protected by the security protocols necessary to simply access the system as a whole and your own filesystem in particular. So, this should still be pretty secure on HPC systems but you might want to use it with caution in less secure environments such as laptops or desktops. For other alternatives, see the documentation on git credentials.

Another action that might make your life easier is to set the following environment variable:

```
export FC=mpifort
```

This is required in order to run with multiple MPI threads within the JEDI CharlieCloud and Singularity containers, which uses OpenMPI. You may wish to put this in a startup-script so you don’t have to enter it manually every time you enter the Container. If you run outside the container, some bundles include customized build scripts that will take care of this for you. Consult the README file in the bundle’s repository for details. If you run `make` and it complains about not finding mpi-related files, try cleaning your build directory (to wipe the CMake cache), setting the FC environment variable as indicated above, and then proceeding with `ecbuild` as described in Step 3 below.

Another thing to keep in mind is that some JEDI tests require six MPI task to run. This is just for ufo-bundle; other bundles may require even more. Chances are good that your machine (whether it be a laptop, a workstation, a cloud computing instance, or whatever), may have fewer than six compute cores.

If your machine has fewer than six compute cores, you may need to explicitly give openmpi permission to run more than one MPI task on each core. To do this, go to the directory `~/.openmpi` (create it if it doesn’t already exist). In that directory, execute this command:

```
echo "rmaps_base_oversubscribe = 1" > mca-params.conf
```

---

**Step 1: Clone the Desired JEDI Bundle**

JEDI applications are organized into high-level **bundles** that conveniently gather together all the repositories necessary for that application to run. Often a bundle is associated with a particular model, such as FV3 or MPAS.

So, to start your JEDI adventure, the first step is to create a directory as a home for your bundle (or bundles--plural--if you’re ambitious!). Here we will use `~/jedi/src` but feel free to call it whatever you wish. Then clone the GitHub repository that contains the bundle you want, as demonstrated here:

```
cd ~/jedi
mkdir src
cd src
git clone https://github.com/JCSDA/ufo-bundle.git
```

---

**Step 2: Choose your Repos**

As executed above, Step 1 will create a directory called `~/jedi/src/ufo-bundle`. `cd` to this directory and have a look (modify this as needed if you used a different path or a different bundle). There’s not much there. There is a README file that you might want to consult for specific information on how to work with this bundle. But in this Step
we’ll focus on the CMakeLists.txt file. This contains a list of repositories that the application needs to run. In the case of ufo-bundle that list looks like this:

```
#ecbuild_bundle( PROJECT eckit GIT "https://github.com/ECMWF/eckit.git" TAG 0.18.5 )
ecbuild_bundle( PROJECT fckit GIT "https://github.com/JCSDA/fckit.git" BRANCH develop UPDATE )
ecbuild_bundle( PROJECT oops GIT "https://github.com/JCSDA/oops.git" BRANCH develop UPDATE )
ecbuild_bundle( PROJECT crtm GIT "https://github.com/JCSDA/crtm.git" BRANCH develop )
ecbuild_bundle( PROJECT ioda GIT "https://github.com/JCSDA/ioda.git" BRANCH develop UPDATE )
ecbuild_bundle( PROJECT ufo GIT "https://github.com/JCSDA/ufo.git" BRANCH develop )
```

Note that the first line is commented out with #. This is because eckit is already installed in the JEDI CharlieCloud and Singularity containers so if you are running inside the container, there is no need to build them again. If you are running outside of the containers and if you have not yet installed these packages on your system, then you may wish to uncomment the line to build eckit. Or, you may wish to install eckit yourself so you can comment this line out in the future. Be warned that can be a bit of a challenge if you are on an HPC system, for example, and you do not have write access to /usr/local. For more information on how to install eckit and ecbuild see our JEDI page on ecbuild and cmake.

As described there, eckit and fckit are software utilities provided by ECMWF that are currently used by JEDI to read configuration files, handle error messages, configure MPI libraries, test Fortran code, call Fortran files from C++, and perform other general tasks. Note that the eckit repository identified is obtained directly from ECMWF whereas the fckit repository is a JCSDA fork. This is because the JEDI team frequently makes changes to fckit so we generally work from our own fork and build it as part of each bundle.

The lines shown above tell ecbuild which specific branches to retrieve from each GitHub repository. **Modify these accordingly if you wish to use different branches.** When you then run ecbuild as described in Step 3 below, it will first check to see if these repositories already exist on your system, within the directory of the bundle you are building. If not, it will clone them from GitHub. Then ecbuild will proceed to checkout the branch specified by the BRANCH argument, fetching it from GitHub if necessary.

If the specified branch of the repository already exists on your system, then ecbuild will not fetch it from GitHub. If you want to make sure that you are using the latest and greatest version of the branch, then there are two things you need to do.

First, you need to include the (optional) UPDATE argument in the ecbuild_bundle() call as shown in each of the lines above. Second, you need to explicitly initiate the update as follows:

```
cd <build-directory>
make update
```

This will tell ecbuild to do a fresh pull of each of the branches that include the UPDATE argument. Note that make update will not work if there is no Makefile in the build directory. So, this command will only work after you have already run ecbuild at least once.

**Warning:** Running make update will initiate a git pull operation for each of the repositories that include the GIT and UPDATE arguments in the call to ecbuild_bundle() in CMakeLists.txt. So, if you have modified these repositories on your local system, there may be merge conflicts that you have to resolve before proceeding.

If you are a developer, you will, by definition, be modifying the code. And, if you are a legitimate JEDI Master, you
will be following the git flow workflow. So, you will have created a feature (or bugfix) branch on your local computer where you are implementing your changes.

For illustration, let’s say we created a feature branch of ufo called feature/newstuff, which exists on your local system. Now we want to tell ecbuild to use this branch to compile the bundle instead of some other remote branch on GitHub. To achieve this, we would change the appropriate line in the CMakeLists.txt file as follows:

```cmake
ecbuild_bundle( PROJECT ufo SOURCE "~/jedi/src/ufo-bundle/ufo" )
```

This will use whatever branch of the specified repository that is currently checked out on your system. As written above, ecbuild will not check out the branch for you. This is usually not a problem because it is likely that you have the appropriate branch checked out already if you are making modifications to it. However, if you do want to insist that ecbuild switch to a particular local branch before compiling, then there is indeed a way to do that:

```cmake
ecbuild_bundle( PROJECT ufo GIT "~/jedi/src/ufo-bundle/ufo" BRANCH feature/newstuff )
```

This may be all you need to know about ecbuild_bundle() but other options are available. For example, if you would like to fetch a particular release of a remote GitHub repository you can do this:

```cmake
ecbuild_bundle( PROJECT eckit GIT "https://github.com/ECMWF/eckit.git" TAG 0.18.5 )
```

For further information see the cmake/ecbuild_bundle.cmake file in ECMWF’s ebuild repository.

**Step 3: Run ecbuild (from the build directory)**

After you have chosen which repositories to build, the next step is to create a build directory (if needed):

```bash
cd ~/jedi
mkdir build
```

Then, from that build directory, run ecbuild, specifying the path to the directory that contains the source code for the bundle you wish to build:

```bash
cd ~/jedi/build
ecbuild ../src/ufo-bundle
```

Here we have used ~/jedi/src as our source directory and ~/jedi/build as our build directory. Feel free to change this as you wish, but just make sure that your source and build directories are different.

This should work for most bundles but if it doesn’t then check in the bundle source directory to see if there are other build scripts you may need to run. This is particularly true if you are running outside of the JEDI CharlieCloud and Singularity containers. These build scripts are customized for each bundle and instructions on how to use them can be found in the README file in the top level of the bundle repository.

**Warning:** Some bundles may require you to run a build script prior to or in lieu of running ecbuild, particularly if you are running outside of the CharlieCloud and Singularity containers. Check the README file in the top directory of the bundle repository to see if this is necessary, particularly if you encounter problems running ecbuild, cmake, or ctest.

After you enter the ecbuild command, remember to practice patience, dear padawan. The build process may take less than a minute for ufo-bundle but for some other bundles it can take twenty minutes or more, particularly if ecbuild has to retrieve a number of large restart files from a remote Git LFS store over a wireless network.
As described *here*, ecbuild is really just a sophisticated (and immensely useful!) interface to CMake. So, if there are any CMake options or arguments you wish to invoke, you can pass them to ecbuild and it will kindly pass them on to CMake. The general calling syntax is:

```
ecbuild [ecbuild-options] [--] [cmake-options] <src-directory>
```

Where `src-directory` is the path to the source code of the bundle you wish to build. The most useful ecbuild option is `debug`:

```
ecbuild --build=debug ../src/ufo-bundle
```

This will invoke the debug flags on the C++ and Fortran compilers and it will also generate other output that may help you track down errors when you run applications and/or tests. You can also specify which compilers you want and you can even add compiler options. For example:

```
ecbuild -- -DCMAKE_CXX_COMPILER=/usr/bin/g++ -DCMAKE_CXX_FLAGS="-Wfloat-equal -Wcast-align" ../src/ufo-bundle
```

Let’s say that you’re working on an HPC system where you do not have the privileges to install Singularity. If this is the case then we recommend that your first check to see if there are JEDI modules installed on your system. If your system is listed on this modules documentation page then you can simply load the JEDI module as described there and you will have access to ecbuild, eckit, and other JEDI infrastructure.

If your system is not one that is supported by the JEDI team, then a second option is to install CharlieCloud in your home directory and run JEDI from within the Charliecloud container.

A third option is for you to install eckit on your system manually (not recommended). If you do this, then you may have to tell ecbuild where to find it with this command line option:

```
ecbuild -- -DECKIT_PATH=<path-to-eckit> ../src/ufo-bundle
```

For more information, enter `ecbuild --help` and see our JEDI page on *ecbuild and cmake*.

**Step 4: Run make (from the build directory)**

Now, at long last, you are ready to compile the code. From the build directory, just type

```
make -j4
```

The `-j4` flag tells make to use four parallel processes. Since many desktops, laptops, and of course HPC systems come with 4 or more compute cores, this can greatly speed up the compile time.

The most useful option you’re likely to want for `make` other than `-j` is the verbose option, which will tell you the actual commands that are being executed in glorious detail:

```
made -j4 VERBOSE=1
```

As usual, to see a list of other options, enter `make --help`.

Again, the compile can take some time (10 minutes or more) so be patient. Then, when it finishes, the next step is to run `ctest`.

### 1.5.2 JEDI Testing

Each JEDI bundle has it’s own suite of tests. To run them, first build and compile the bundle as described in our *bundle build page*. Step 5 in that building and compiling procedure is to test the code with `ctest`. This step is described in the following section.
After describing the basic functionality of ctest, we proceed to give a more detailed overview of how tests are organized and implemented in JEDI. This is a prelude to the next document, which describes how you — yes you! — can implement your own JEDI unit tests.

**Running ctest**

The standard practice after building and compiling a JEDI bundle is to run ctest with no arguments in order to see if the bundle is operating correctly:

```
cd <build-directory>
ctest
```

This will run all tests in the test suite for that bundle. This can take a while so be patient. When the tests are complete, ctest will print out a summary, highlighting which tests, if any, failed. For example:

```
98% tests passed, 2 tests failed out of 130

Label Time Summary:
boost = 2253.85 sec (75 tests)
executable = 2253.85 sec (75 tests)
fv3jedi = 2241.67 sec (10 tests)
mpi = 2242.21 sec (11 tests)
oops = 28.63 sec (111 tests)
script = 26.18 sec (55 tests)
ufo = 9.73 sec (9 tests)

Total Test time (real) = 2280.20 sec

The following tests FAILED:
  123 - test_fv3jedi_aninit (Failed)
  130 - test_fv3jedi_localization (Timeout)

Errors while running CTest
```

If you want to run a single test or a subset of tests, you can do this with the -R option, for example:

```
cctest -R test_fv3jedi_linearmodel # run a single test
cctest -R test_qg* # run a subset of tests
```

The output from these tests (stdout) will be printed to the screen but, to allow for greater scrutiny, it will also be written to the file `LastTest.log` in the directory `<build-directory>/Testing/Temporary`. In that same directory you will also find a file called `LastTestsFailed.log` that lists the last tests that failed. This may be from the last time you ran ctest or, if all those tests passed, it may be from a previous invocation.

If you’re not happy with the information in LastTest.log and you want to know more, you can ask ctest to be *verbose*

```
cctest -V -R test_fv3jedi_linearmodel
```

...or even *extra-verbose* (hypercaffeinated mode):

```
cctest -VV -R test_fv3jedi_linearmodel
```

The -V and even -VV display the output messages on the screen in addition to writing them to the LastTest.log file. However, sometimes the amount of information written to LastTest.log isn’t much different than if you were to run ctest without these options, particularly if all the tests pass.

Another way to get more information is to set one or both of these environment variables before you run ctest:
The first enables debug messages within the JEDI code that would not otherwise be written. The second produces messages that follow the progress of the code as it executes. Both tools are provided by `eckit`. Though higher values of these variables could in principle be set, few JEDI routines exploit this functionality. So, setting these variables to values greater than 1 will make little difference. Both can be disabled by setting them to zero.

`ctest` also has an option to only re-run the tests that failed last time:

```
ctest --rerun-failed
```

To see a list of tests for your bundle without running them, enter

```
ctest -N
```

For a complete list of ctest options, enter `man ctest`, `ctest --help`, or check out our [JEDI page on CMake and CTest](#). As described there, CTest is a component of CMake, so you can also consult the CMake online documentation for the most comprehensive documentation available.

### Manual Execution

You can also run the executable test files directly, without going through ctest. To do this, first find the executable in the build directory. Unit tests are typically found in one of the `test` directories that branch off each repository name. For example, `test_qg_state` can be found in `<build-directory>/oops/qg/test` and `test_ufo_geovals` can be found in `<build-directory>/ufo/test`. Then just `cd` to that directory and run the executable from the command line, specifying the appropriate input (configuration) file, e.g.

```
test_qg_state testinput/interfaces.yaml
```

You can determine which executable and which configuration file each test uses by viewing the `CMakeLists.txt` file in the corresponding `test` directory of the repository. If you’re running the ufo bundle, then the relevant `CMakeLists.txt` files for the examples above would be `<src-directory>/ufo-bundle/oops/qg/test` and `<src-directory>/ufo-bundle/ufo/test`. Just open the relevant `CMakeLists.txt` file and search on the name of the test. See [Adding a New Test](#) for further details on how to interpret the syntax.

If you do run the tests without ctest, keep in mind a few tips. First, the test name is not always the same as the executable name. Second, since the the integration and system tests generally focus on JEDI Applications (other than `oops::Test` objects - see below) they usually have a `.x` extension. Furthermore, these executables are generally located in the `<build-directory>/bin` directory as opposed to the `test` directories. For example, to run `test_qg_truth` from the `<build-directory>/oops/qg/test` directory, you would enter the following:

```
../../../bin/qg_forecast.x testinput/truth.yaml
```

### The JEDI test suite

What lies “under the hood” when you run `ctest`? Currently, there are two types of tests implemented in JEDI:

1. Unit tests
2. Integration and system tests (aka Application tests)

This does not include other types of system and acceptance testing that may be run outside of the ctest framework, by individual developers and testers. Integration and system tests are refereed to as Application tests for reasons that will become clear in the next section.
Unit tests are currently implemented in JEDI using eckit unit testing framework for initializing and organizing our suite of unit tests, and for generating appropriate status and error messages. See below for further details on how tests are implemented.

Unit testing generally involves evaluating one or more Boolean expressions during the execution of some particular component or components of the code. For example, one can read in a model state from an input file and then check whether some measure of the State norm agrees with a known value to within some specified tolerance. Or, one can test whether a particular variable is positive (such as temperature or density) or whether a particular function executes without an error.

By contrast, Application tests check the operation of some application as a whole. Some may make use of eckit Boolean tests but most focus on the output that these applications generate. For example, one may wish to run a 4-day forecast with a particular model and initial condition and then check to see that the result of the forecast matches a well-established solution. This is currently done by writing the output of the test to a file (typically a text file) and comparing it to an analogous output file from a previous execution of the test. Such reference files are included in many JEDI repositories and can generally be found in a test/testoutput subdirectory.

Comparisons between output files are currently done by means of the compare.sh bash script which can be found in the test subdirectory in many JEDI repositories. This script uses standard unix parsing commands such as grep and diff to assess whether the two solutions match. For further details see the section on Integration and System testing below.

Warning: The compare.sh testing procedure is provisional and is likely to be modified in the future.

As mentioned above, each JEDI bundle has its own suite of tests and you can list them (without running them) by entering this from the build directory:

```
ctest -N
```

Though all tests in a bundle are part of the same master suite, they are defined within each of the bundle’s individual repositories. Furthermore, you can generally determine where each test is defined by its name. For example, all test_qg_* tests are defined in oops/qg; all test_ufo_* tests are defined in ufo; all test_fv3jedi_* tests are defined in the fv3-jedi repo, and so on.

With few exceptions, all JEDI repositories contain a test directory that defines the tests associated with that repository. oops itself is one exception because it orchestrates the operation of the code as a whole but there you will find archetypical test directories within the qg and 195 model directories.

Within each test directory you will find a file called CMakeLists.txt. This is where each test is added, one by one, to the suite of tests that is executed by ctest. As described in the CMake documentation, this is ultimately achieved by repeated calls to the CMake add_test() command.

However, the ecbuild package offers a convenient interface to CMake’s add_test() command called ecbuild_add_test(). Application tests are added by specifying TYPE SCRIPT and COMMAND "compare.sh" to ecbuild_add_test(). For further details on how to interpret this argument list see Adding a New Unit Test.

Since it relies on the net result of an application, each Application test is typically associated with a single ctest executable. However, applications of type oops::Test (see next section) will typically execute multiple unit tests for each executable, or in other words each item in the ctest suite. So, in this sense, the suite of unit tests is nested within each of the individual tests defined by ctest. And, it is this nested suite of unit tests. (see below).

Tests as Applications

The JEDI philosophy is to exploit high-level abstraction in order to promote code flexibility, portability, functionality, efficiency, and elegance. This abstraction is achieved through object-oriented design principles.
As such, the execution of the JEDI code is achieved by means of an Application object class that is defined in the oops namespace. As illustrated in the following class hierarchy, oops::Test is a sub-class of the oops::Application class, along with other applications such as individual or ensemble forecasts:

Unit tests are implemented through oops::Test objects as described in this and the following sections. The other type of test in the JEDI test suite, namely Application tests, generally check the operation of JEDI applications as a whole - the same applications that are used for production runs and operational forecasting. In other words, application tests are used to test the operation of the Application classes in the diagram above that are not sub-classes of oops::Test.

To appreciate how a JEDI Application is actually run, consider the following program, which represents the entire (functional) content of the file oops/qg/test/executables/TestState.cc:

```cpp
int main(int argc, char ** argv) {
    oops::Run run(argc, argv);
    test::State<qg::QgTraits> tests;
}
```

(continues on next page)
run.execute(tests);
    return 0;
};

This program begins by defining an object of type `oops::Run`, passing the constructor the arguments from the command line. These command-line arguments generally include a configuration file that specifies the parameters, input files, and other information that is necessary to run the application (in this case, a test).

Then the program proceeds to define an object of type `test::State<qg::QgTraits>` called `tests`, which is a sub-class of `oops::Test` as illustrated here:
Since `test::State<qg::QgTraits>` is a sub-class of `oops::Test` (through the appropriate instantiation of the `test::State<MODEL>` template), then the `tests` object is also an Application (`oops::Application`).

So, after defining each of the objects, the program above proceeds to pass the Application object (`tests`) to the `execute()` method of the `oops::Run` object. Other applications are executed in a similar way.

Source code for the executable unit tests in a given JEDI repository can typically be found in a sub-directory labelled `test/executables` or `test/mains`. Similarly, the source code for executable JEDI Applications that are not `oops::Test` objects can typically be found in a `mains` directory that branches from the top level of the repository.

### Initialization and Execution of Unit Tests

As described *above*, an `oops::Test` object is an application that is passed to the `execute()` method in an `oops::Run` object. To describe what happens next, we will continue to focus on the `test_qg_state` example introduced in the previous section as a representative example.

First, it is important to realize that the `test::State<Model>` class is not the same as the `oops::State<Model>` class. The former is an application as described in the previous section whereas the latter contains information about and operations on the current model state.

Second, as an application, a `test::State<Model>` object also has an `execute()` method, which is called by the `execute()` method of the `oops::Run` object as shown here (code excerpt from `oops/src/oops/runs/Run.cc`):

```cpp
void Run::execute(const Application & app) {
    int status = 1;
    Log::info() << "Run: Starting " << app << std::endl;
    try {
        status = app.execute(*config_);
    }
    [...]
}
```

The `execute()` method for an `oops::Test` is defined in the file `oops/src/oops/runs/Test.h`. The main purpose of this routine is to initialize and run the suite of unit tests.

The `execute()` method in each `oops::Test` object then proceeds to register the tests with `oops::Test::register_tests()` and run them with a call to eckit’s `run_tests()` function (`argc` and `argv` are parsed from the `args` variable above):

```cpp
// Run the tests
Log::trace() << "Registering the unit tests" << std::endl;
register_tests();
Log::trace() << "Running the unit tests" << std::endl;
int result = eckit::testing::run_tests(argc, argv, false);
Log::trace() << "Finished running the unit tests" << std::endl;
Log::error() << "Finished running the unit tests, result = " << result << std::endl;
```

So, the real difference between different `oops::Test` objects is encapsulated in the `oops::Test::register_tests()` method. Each test application (i.e. each item in ctest’s list of tests) will register a different suite of unit tests.

In the case of `test::State<MODEL>` (which you may recall from the previous section is a sub-class of `oops::Test`), this method is defined as follows (see `oops/src/test/interface/State.h`):

```cpp
void register_tests() const {
    std::vector<eckit::testing::Test>& ts = eckit::testing::specification();
}
```

(continues on next page)
This is where the eckit unit test suite is actually initiated: A \texttt{ts} object is created by calling \texttt{specification()}, tests are added to testing suite \texttt{ts} by \texttt{emplace_back}.

Note that all this occurs within the \texttt{test::State<MODEL>} class template so there will be a different instance of each of these unit tests for each model. So, our example application \texttt{test\_qg\_state} will call \texttt{test::State<qg:QgTraits>::register\_tests()} whereas other models and other applications (as defined in other sub-classes of \texttt{oops::Test} - see \textit{above}) will register different unit tests.

So, in short, members of the \texttt{ctest} test suite are added by means of \texttt{ecbuild\_add\_test()} commands in the appropriate \texttt{CMakeLists.txt} file (see \textit{above}) while members of the nested unit test suite are added by means of the \texttt{oops::Test::register\_tests()} method.

Anatomy of a Unit Test

Let’s continue to use \texttt{test\_qg\_state} as an example in order to illustrate how unit tests are currently implemented in JEDI. As described in the previous two sections, the execution of this test (a single test from the perspective of \texttt{ctest}) will call \texttt{test::State<qg:QgTraits>::register\_tests()} to register a suite of unit tests and it will call \texttt{eckit::testing::run\_tests()} to run them.

As demonstrated in the previous section, this particular suite of unit tests includes two members, namely \texttt{testStateConstructors<MODEL>()} and \texttt{TestStateInterpolation<MODEL>()}, with \texttt{MODEL} instantiated as \texttt{qg:QgTraits}. What happens when we run one of these unit tests?

Here we will focus on the first, \texttt{TestStateConstructors<MODEL>()}. Both are defined in \texttt{oops/src/test/interface/State.h}, where you will find this code segment:

```cpp
template <typename MODEL> void testStateConstructors() {
  typedef StateFixture<MODEL> Test_;
  typedef oops::State<MODEL> State_;

  const double norm = Test_::test().getDouble("norm-file");
  const double tol = Test_::test().getDouble("tolerance");
  const util::DateTime vt(Test_::test().getString("date"));

  // Test main constructor
  const eckit::LocalConfiguration conf(Test_::test(), "StateFile");
  const oops::Variables vars(conf);
  boost::scoped_ptr<State_> xx1(new State_(Test_::resol(), vars, conf));

  EXPECT(xx1.get());
  const double norm1 = xx1->norm();
  EXPECT(oops::is_close(norm1, norm, tol));
  EXPECT(xx1->validTime() == vt);
  
  [...]  
```

This starts by defining \texttt{Test\_} as an alias for the \texttt{StateFixture<MODEL>} class. Other test objects also have corresponding fixture classes, for example \texttt{test::ModelFixture<MODEL>}, \texttt{test::ObTestsFixture<MODEL>}, etc. These are primarily used to access relevant sections of the configuration file. In the above example, they are used to extract a reference value for the State norm, a tolerance level for
the norm test, and a reference date for the State object that is about to be created.

Then the “StateFile” section of the config file is extracted through the StateFixture and, together with information about the geometry (in Test_::resol()), is used to create a new State object called *xx1 (boost::scoped_ptr<> is a type of smart pointer defined by Boost similar to std::unique_ptr<> in C++11).

Then the unit tests really begin, with multiple calls to check Boolean expressions, including exit codes. The first call to EXPECT() checks to see if the pointer is properly defined with the help of the get() method of boost::scoped_ptr<> In other words, it checks to see if a State object was successfully created.

The call to EXPECT(oops::is_close(norm1, norm, tol)) then checks to see if the norm that was read from the configuration file is equal to the value computed with the norm() method of the State object, with the specified tolerance.

EXPECT() with double equal sign is used to verify that the State object is equal to the reference value read from the configuration file.

The function above then proceeds to perform similar tests for the copy constructor (not shown).

If any of these nested unit tests fail, ctest registers a failure for the parent application and an appropriate message is written to the ctest log file (as well as stdout if ctest is run in verbose mode).

**Integration and System (Application) Testing**

Though each executable in the ctest test suite may run a number of unit tests as described in the previous two sections, many are also used for higher-level integration and system testing. As described in *The JEDI Test Suite* above, these are currently implemented by comparing the output of these executables to known solutions.

Files containing summary data for these known solutions can be found in the test/testoutput directory of many JEDI reposistories. The test_qg_state example that we have been using throughout this document is a unit test suite (Type 1) as opposed to an Application test (Type 2) so it does not have a reference output file. However, as an Application test, test_qg_truth does have such a file. The name of this reference file is truth.test and its contents are as follows:

<table>
<thead>
<tr>
<th>Test</th>
<th>Initial state: 13.1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Test</td>
<td>Final state: 15.1417</td>
</tr>
</tbody>
</table>

This lists the norm of the initial and final states in an 18 day forecast. So, the ostensibly sparse contents of this file are misleading: a lot of things have to go right in order for those two data points to agree precisely!

This and other reference files are included in the GitHub repositories but the output files themeselves are not. They are generated in the build directory by running the test, and they follow the same directory structure as the repository itself. Furthermore, they have the same name as the reference files they are to be compared to but with an extension of .test.out. Messages sent to stdout during the execution of the test are written to another file with an extension .log.out.

So, in our example above, the output of test_qg_truth will be written to

```
<build-directory>/oops/qg/test/testoutput/truth.test.test.out`
```

In the same directory you will find a soft link to the reference file, truth.test, as well as the log file, truth.test.log.out.

When the test is executed, the compare.sh script in the test directory of the repository (which also has a soft link in the build directory) will compare the output file to the reference file by first extracting the lines that begin with “Test” (using grep) and then comparing the (text) results (using diff). In our example, the two files to be compared are test.truth and test.truth.test.out. If these do not match, ctest registers a failure.
Warning: The compare.sh script may have problems if you run with multiple processors.

JEDI Testing Framework

In this document we have described how unit tests are implemented as oops::Test (Application) objects and we have described how they are executed by passing these Application objects to an oops::Run object. We have focused on the oops repository where this testing framework is currently most mature. However, the ultimate objective is to replicate this structure for all JEDI repositories.

Using oops as a model, the objective is to have the test directory in each JEDI repository mirror the src directory. So, ideally, every class that is defined in the src directory will have a corresponding test in the test directory. Furthermore, each of these tests is really a suite of unit tests as described above.

Let’s consider ufo as an example. Here the main source code is located in ufo/src/ufo. In particular, the .h and .cc files in this directory define the classes that are central to the operation of ufo. For each of these classes, there should be a corresponding .h file in ufo/test/ufo that defines the unit test suite for objects of that class. These are not yet all in place, but this is what we are working toward. The same applies to all other JEDI repositories.

Each unit test suite should be defined as a sub-class of oops::Test as described above. Then it can be passed to an oops::Run object as an application to be executed.

For further details on how developers can contribute to achieving this vision, please see Adding a New Test.

1.5.3 Adding a New Test

So, you’ve developed a new feature in some JEDI repository and now you want to test it. You have come to the right place.

This document is a step-by-step guide on how to add a test to JEDI. Here we focus on unit tests but in the final section we include some guidance on how you might proceed to Add an Application test.

The first thing to emphasize is that there are many levels of sophistication on how you might wish to proceed. The simplest case is to use the existing unit test infrastructure. If you go this route, then you may just have to create an appropriate configuration file and then register your test with CMake and CTest (Steps 7-8). Or, you may wish to add a new test to an existing test suite, to create a new test suite, or even to establish the proper directory structure.

Wherever you are on this spectrum of possibilities, we hope this document will be useful. Just be aware that you are under no obligation to follow all the steps. If the infrastructure for that step is already in place, then feel free to proceed to the next step.

In any case, it is imperative that you first read this document that describes how tests are organized, implemented and executed within JEDI. In particular, please read our vision on how we would like the development of the JEDI testing framework to proceed.

Step 1: Create a File for your Test Application

The goal of the JEDI Testing Framework is to have the test directory mirror the source directory such that each of the main C++ classes defined in the source directory has a corresponding test.

So let’s say that there is a file in some JEDI repository called src/mydir/MyClass.h that defines a C++ class called MyClass. And, let’s say that we want to define a unit test suite to test the various methods and constructors in MyClass. The first thing we would want to do is to create a file called test/mydir/MyClass.h that will contain the test application.
If you’re working in a well-established JEDI directory then this file may already exist. If that’s the case, then you can probably move on to Step 3. On the other hand, if you’re adding a new model to JEDI, it’s possible that the directory won’t exist yet, let alone the file. So, create the directory and the file as needed.

Before proceeding we should emphasize again that there is another option. Often the existing test applications defined within `oops/src/test/interface` will be sufficient to test your new feature. If this is the case, then you can skip ahead to Step 6. There you will see an example where this is done for FV3.

Or, even better - sometimes the test infrastructure will already exist within the JEDI repository you are working with. Often you can reuse an existing test and just specify a different configuration that activates your new feature. In that case all you have to do is to create an appropriate configuration file and you can proceed directly to Step 7!

If you’re still in this section, read on for some tips on how to write the application file `test/mydir/MyClass.h`.

**What MyClass.h Should Contain**

As a prelude to the steps that follow, we note that the main purpose of `test/mydir/MyClass.h` is to define `test::MyClass` as a sub-class of `oops::Test`. As a sub-class of `oops::Test`, `test::MyClass` will also be a sub-class of `oops::Application`. That is, it will be an Application object that can be executed by *passing it to an oops::Run object*. In addition to declaring and defining `test::MyClass`, our file might also define a `MyClassFixture` class to help with accessing the configuration file (Step 2). Necessary components of `test/mydir/MyClass.h` include one or more functions that define the unit tests as well as a `register_tests()` method within `test::MyClass` that adds these tests to the master test suite.

Since we’ll be building off of `oops::Test` and the eckit unit test suite, one necessary item in the header of our MyClass.h file is:

```plaintext
#define ECKIT_TESTING_SELF_REGISTER_CASES 0
```

And the contents of the file should be encapsulated within the `test` namespace, to distinguish it from the corresponding class of the same name in the `src` directory.

**What MyClass.h Should not Contain**

As described elsewhere, the unit tests are initialized and executed by means of the `execute()` method in `oops::Test`. As a sub-class of `oops::Test`, the `test::MyClass` object will have access to this method and it is best not to re-define it. Doing so may disrupt how eckit executes the tests.

**Step 2: Define A Test Fixture**

In JEDI, test fixtures are generally used to create objects as directed by the relevant sections of the *configuration file*, for use with the unit tests. As an example, consider this code segment in `oops/src/test/interface/Increment.h`:

```plaintext
template <typename> class IncrementFixture : private boost::noncopyable {
    typedef oops::Geometry<MODEL> Geometry_;

    public:
        static const Geometry_ & resol() { return getInstance().resol_; }
        static const oops::Variables & ctlvars() { return getInstance().ctlvars_; }
        static const util::DateTime & time() { return getInstance().time_; }

    private:
        static IncrementFixture<MODEL> & getInstance() {
```

(continues on next page)
Note that this, like other oops test objects, is a class template, with a different instance for each model. This may not be necessary for your test if it is model-specific. The main point here is that the \texttt{resol()}\texttt{, ctlvars()}\texttt{, and time()} methods of \texttt{test::IncrementFixture<MODEL> access the “Geometry”, “Variables”, and “TestDate” sections of the configuration file and use this information to create objects of type oops::Geometry<MODEL>\texttt{, oops::Variables, and util::DateTime. These methods are then used repeatedly by the various unit tests that are included in test::Increment. The TestEnvironment::config() calls in the code above provide a way to pass global configuration data to the test fixtures.}

So, proceeding with our example, it would be advisable to begin by defining a \texttt{test::MyClassFixture} class in \texttt{test/mydir/MyClass.h} to facilitate the creation of useful objects as specified in the configuration file. For many more examples see the various files in \texttt{oops/src/test/interface}.

**Step 3: Define Your Unit Tests**

Now the next step would be to define the unit tests themselves as functions within \texttt{test/mydir/MyClass.h}. As a guide you can use the illustrative example in \texttt{Anatomy of a Unit Test} or the many examples to be found in \texttt{oops/src/test/interface}. The possibilities are endless, but just remember two things:

1. Include one or more calls to \texttt{eckit check functions}
2. Use your test fixture to create objects based on the information in the configuration file

**Step 4: Register your Unit Tests with eckit**

In order for eckit to run your tests, you have to \texttt{register} each individual test. This is achieved by means of the \texttt{register_tests()} method of \texttt{test::MyClass} and as this \texttt{test::Increment} example (from \texttt{oops/src/test/interface/Increment.h}) demonstrates, there is little else needed to define the class:
private:
std::string testid() const {return "test::Increment" + MODEL::name() + ">";}

void register_tests() const {
    std::vector<eckit::testing::Test>& ts = eckit::testing::specification();
    ts.emplace_back(CASE("interface/Increment/testIncrementConstructor")
        {testIncrementConstructor<MODEL>(); });
    ts.emplace_back(CASE("interface/Increment/testIncrementCopyConstructor")
        {testIncrementCopyConstructor<MODEL>(); });
    ts.emplace_back(CASE("interface/Increment/testIncrementTriangle")
        {testIncrementTriangle<MODEL>(); });
    ts.emplace_back(CASE("interface/Increment/testIncrementOpPlusEq")
        {testIncrementOpPlusEq<MODEL>(); });
    ts.emplace_back(CASE("interface/Increment/testIncrementDotProduct")
        {testIncrementDotProduct<MODEL>(); });
    ts.emplace_back(CASE("interface/Increment/testIncrementAxpy")
        {testIncrementAxpy<MODEL>(); });
    ts.emplace_back(CASE("interface/Increment/testIncrementInterpAD")
        {testIncrementInterpAD<MODEL>(); });
}

So, we would proceed by defining test::MyClass in a similar way. Just specify the test object (here ts) and add each of your test functions one by one using emplace_back as shown.

Then no more action is required for test/mydir/MyClass.h; Our test::MyClass::register_tests() method will be executed automatically when we pass test::MyClass as an application to oops::Run (see Initialization and Execution of Unit Tests).

**Step 6: Create an Executable**

Executables for each test are generally located in the test/executables directory of each JEDI repository, though sometimes this directory is called test/mains. This is not to be confused with the mains directory (branching off the top level of the repository) which is typically reserved for the production-level programs.

As described in Tests as Applications, there is not much for the executable file to do. It only really has three tasks:

1. Create an oops::Run object
2. Create an oops::Application object (in our example, this would be test::MyClass)
3. Pass the Application object to the execute() method of the Run object

So, to proceed with our example, we might go to the test/executables directory of our repository (create it if it’s not there already) and create a file called TestMyClass.cc with the following contents:

```cpp
#include "oops/runs/Run.h"
#include "/mydir/MyClass.h"

int main(int argc, char ** argv) {
    oops::Run run(argc, argv);
    test::MyClass tests;
    run.execute(tests);
    return 0;
}
```
That’s it. Note that the include paths for a given repository are specified in the CMakeLists.txt file in the top level of the repository. All existing JEDI repositories will already have access to `oops/src` by means of these lines, or something similar:

```cmake
ecbuild_use_package( PROJECT oops VERSION 0.2.1 REQUIRED )
include_directories( ${OOPS_INCLUDE_DIRS} )
```

So, the first include statement in the TestMyClass.cc example above should have no problem finding `oops/src/oops/runs/Run.h`, where the `oops::Run` class is defined.

It is likely that the `src` directory of the working repository is also in the include path. So, in the above example we specified the relative path of our `MyClass.h` file in the `test` directory so the compiler does not confuse it with the file of the same name in the `src` directory.

In some situations it might be beneficial to define a modified Run object that does some additional model-specific set up. Here is an example from `fv3-jedi/test/executables/TestModel.cc`

```cpp
#include "FV3JEDITraits.h"
#include "RunFV3JEDI.h"
#include "test/interface/Model.h"

int main(int argc, char ** argv) {
    fv3jedi::RunFV3JEDI run(argc, argv);
    test::Model<fv3jedi::FV3JEDITraits> tests;
    run.execute(tests);
    return 0;
}
```

However, `fv3jedi::RunFV3JEDI` is a sub-class of `oops::Run` and it uses the `execute()` method of its parent. So, the execution of the test is essentially the same as the previous example.

Also, it is worth noting that the application used here is the `fv3jedi::FV3JEDITraits` instance of `test::Model<MODEL>`, which is already defined in `oops/src/test/interface/Model.h`. So, in this case there would be no need to create a new test application as described in Steps 1-5.

### Step 7: Create a Configuration File

Along with the executable, the **configuration file** is the way to tell JEDI what you want it to do. We reserve a detailed description of how to work with JEDI configuration files for another document.

Here we’ll just say that the proper place to put it is in the `test/testinput` directory of the JEDI repository that you are working with. Or, if your tests are located in `test/mydir`, another option would be to put the associated input files in `test/mydir/testinput`. If there are already some files there, you can use them as a template for creating your own. Or, you can look for `testinput` files from other repositories that test similar functionality.

Let’s call our configuration file `test/testinput/myclass.yaml`. To proceed, we would create the file and then edit it to activate the code features that we wish to test.

As mentioned way back in Step 1, some tests do not require new infrastructure. Some new tests only require a different configuration file to activate a different feature of the code. If this is the case for you, then you can just duplicate an existing configuration file and modify it accordingly, skipping Steps 1-6.

### Step 8: Register all files with CMake and CTest

In steps 1-7 above we have created or modified three files, namely the source code for our tests, `test/mydir/MyClass.h`, the executable `test/executables/TestMyClass.cc`, and the configuration file `test/
In order for CMake to compile and run these files, we have to let CMake know they exist.

We achieve this by editing the file `test/CMakeLists.txt`. This is where the tests are managed from the perspective of CMake and CTest.

We’ll start with the configuration file because every new test you add is likely to have a new configuration file. Edit the `CMakeLists.txt` file and look for a list of input files like this one from `oops/qg/test/CMakeLists.txt`:

```bash
list( APPEND qg_test_input
testinput/3dvar.yaml
testinput/3dfgat.yaml
testinput/4densvar.yaml
testinput/4dvar.alpha.yaml
[...]
testinput/test_op_obs.yaml
testinput/analytic_init.yaml
testinput/analytic_init_fc.yaml
compare.sh
)
```

You would add your input file, `test/testinput/myclass.yaml` to this list (note that the path is relative to the path of the `CMakeLists.txt` file itself). If you search on `qg_test_input` in the file, you can see that list is later used to create a soft link for the input files in the build directory, where the tests will be run.

Finally, at long last, you can register your test with CTest. We can do this with a call to `ecbuild_add_test()` in the `test/CMakeLists.txt` file. Here is an example from `oops/qg/test/CMakeLists.txt`:

```bash
ecbuild_add_test( TARGET test_qg_state
  SOURCES executables/TestState.cc
  ARGS "testinput/interfaces.yaml"
  LIBS qg )
```

The TARGET option defines the name of the test. The use of TARGET, as opposed to COMMAND, tells CMake to compile the executable before running it. This requires that we specify the executable with the SOURCES argument, as shown.

The configuration file is specified using the ARGS argument to `ecbuild_add_test()`. This will be implemented as a command-line argument to the executable as described in Manual Execution. The LIBS argument specifies the relevant source code through a previous call to `ecbuild_add_library()`.

So, our example would look something like this:

```bash
ecbuild_add_test( TARGET test_myrepo_myclass
  SOURCES executables/TestMyClass.cc
  ARGS "../testinput/myclass.yaml"
  LIBS myrepo )
```

Note that this is sufficient to inform CMake of the existence of our executable so it need not appear in any other list of files (such as `test_qg_input` above or similar lists of source files used to create the ecbuild libraries). Furthermore, since the executable includes our test application file `test/mydir/MyClass.h`, it will be compiled as well, as part of the compilation of the executable. So, we’re done! Good luck with debugging!

There are many other useful arguments for `ecbuild_add_test()`. As usual, the best source for information is the file that defines the macro itself, `cmake/ecbuild_add_test.cmake` in ECMWF’s ecbuild repository. And, as usual, we recommend that you peruse the other JEDI repositories for relevant examples. If you want to add input data files and/or Fortran namelists to your test configurations, have a look at how this is done in `fv3-jedi/test/CMakeLists.txt`.

1.5. Building, Testing, and Running JEDI
Adding an Application Test

The steps above are specific to Unit Tests. You could in principle follow much of the same procedure to create an Application test but since these are usually used to test existing Applications, steps 1-5 would usually not be necessary.

You would have to design your application to produce a text output file as described in Application Testing and you would have to provide a reference output file to compare against. These reference output files would be have to be added to the CMakeLists.txt file in much the same way as the input configuration files (Step 8) in order to ensure that they will be visible from the build directory; see oops/qg/test/CMakeLists.txt for an example.

You would add your test to the appropriate CMakeLists.txt file with ecbuild_add_test() as described in Step 8 but the argument list would be somewhat different as illustrated here:

```bash
ecbuild_add_test( TARGET test_qg_truth
  TYPE SCRIPT
  COMMAND "compare.sh"
  ARGS "$(CMAKE_BINARY_DIR)/bin/qg_forecast.x testinput/truth.yaml"
  testoutput/truth.test
  DEPENDS qg_forecast.x )
```

Here we include a TYPE SCRIPT argument and we specify command.sh as the command to be executed. The ARGS argument now includes the two files to be compared, namely the output of our application `$(CMAKE_BINARY_DIR)/bin/qg_forecast.x testinput/truth.yaml` (set off by quotes) and our reference file, testoutput/truth.test. We include the executable application in the DEPENDS argument to make sure that CMake knows it needs to compile this application before running the test.

However, before you add an Application test we must warn you again that the compare.sh script may run into problems if you run your application on multiple MPI threads. We are currently working on a more robust framework for Application testing.

1.5.4 JEDI Configuration Files: Implementation

This document describes the practical implementation of JEDI configuration files, including how users can create and read them. For an overview what these files contain see JEDI Configuration Files: Content.

File Format

Configuration files are files that control the execution of specific applications. They specify input parameters, control flags, file names, tolerance thresholds, and other configuration details that are used by JEDI to run tests, forecasts, DA, and other applications.

Configuration files for most JEDI tests are located in the test/testinput directory of each JEDI repository. The configuration files in the ufo, ioda, and fv3-jedi repositories are particularly useful as illustrative examples for constructing configuration files in other repositories.

Configuration (often abbreviated as config) files in JEDI may be written either in YAML or in JSON; JEDI uses the parser from ECMWF’s eckit library to read these files and this parser can process both formats. However, we recommend using YAML because it is generally more user-friendly; YAML is easier to read and, unlike JSON, it allows the user to include comments. YAML files can end with the extension .yaml or .yml. JSON files typically end with the extension .json.

As an example, consider the configuration file for the test_ufo_radiosonde_opr unit test (also used for several other tests), which is located in the ufo repository as test/testinput/radiosonde.yaml:
Note that keys representing single variables or vectors are represented as lower case while keys representing more complex objects in the YAML hierarchy are rendered in CamelCase. This is the preferred style but it is not currently followed by all JEDI repositories.

We refer the user to the YAML Documentation for a comprehensive description of the syntax but we’ll give a brief overview here.

The first thing to note is that indentation matters. Items are organized into a hierarchy, with the top-level objects beginning in the leftmost column and subsidiary components of these objects indented accordingly. The number of spaces is not important; two is sufficient to define the scope of an item and its contents.

The beginning of a YAML document is indicated by three dashes ---, which may or may not be preceded by directives. Each line typically contains a key-value pair separated by a colon and a space. The key is generally a string and the value may be either a string or a number. This is used to assign values to variables. For example, the window_begin object is set to a value of ‘2018-04-14T21:00:00Z’ and the LinearObsOpTest.toleranceTL variable is set to a value of 1.0e-9. Note that we have used a period to represent the hierarchy of items; toleranceTL is a component of LinearObsOpTest. Note also that the values may be interpreted in different ways. For example, the window_begin value is written as a string in the yaml file but it is interpreted as a util::DateTime object when it is read into JEDI.

Objects with multiple values (sequences in YAML) are indicated as indented lists with one item per line and each item delineated by a dash. For example, Observations.ObsTypes[0].variables is equated to a list of items, namely [“air_temperature”, “eastward_wind”, “northward_wind”]. Comments are preceded by a # sign as seen for Observations.ObsTypes[0].tolerance.
Lists or sequences may also be identified with brackets \([\])\. This is illustrated in the above file with the example of `Observations.ObsTypes[0].ObsBias` which is here identified as a list, albeit an empty one.

**C++ Usage**

As noted in the previous section, JEDI configuration files are read by means of the `eckit` C++ library developed and distributed by the European Centre for Medium Range Weather Forecasting (ECMWF).

Configuration files are read into JEDI as `eckit::Configuration` objects. More specifically, `eckit::Configuration` is the base class that is often accessed through its derived classes `eckit::LocalConfiguration` and `eckit::YAMLConfiguration`. All of these classes are defined in the `src/eckit/config` directory of the `eckit` repository.

As described in our document on *JEDI Testing* (see *Tests as Applications* in particular), JEDI applications are executed by passing an `oops::Application` object to the `execute()` method of an `oops::Run` object. The name of the configuration file (including path) is generally specified on the command line when running a JEDI executable and this file name is passed to the constructor of the `oops::Run` object. There is it used to create an `eckit::Configuration` object which is passed to the Application when it is executed. The `eckit::Configuration` class contains a number of public methods that can be then used to query the config file and access its contents.

To illustrate how this works, let’s return to our `test_ufo_radiosonde_opr` example introduced in the previous section. The configuration file for that test is called `test/testinput/radiosonde.yaml`. In this example, our Application happens to be a Test object since `oops::Test` is a subclass (child) of `oops::Application`. So, the configuration file is passed from the command line to the `oops::Run` object and then to the Application as an argument (of type `eckit::Configuration`) to the `oops::Test::execute()` method. This general approach is similar to other Applications.

What happens next is more specific to the Test Application but it serves to illustrate how to manipulate and access the config file as an `eckit::Configuration` object. Here is a code segment from the `oops::Test::execute()` method as defined in the `oops/src/oops/runs/Test.h` file:

```cpp
int Test::execute(const eckit::Configuration & config) const {

    // Setup configuration for tests
    test::TestEnvironment::getInstance().setup(config);

    // Extract the runtime config for the tests from the config file.
    std::string args = config.getString("test_framework_runtime_config");

    // ...
}
```

Here the Configuration object that is passed as an argument (config) is used to create and initialize a `TestEnvironment` object. This is used later to facilitate access to the config file for the test suite as we will see below. However, the config file can also be accessed directly through the public methods of the `eckit::Configuration` object itself. This is demonstrated by the `config.getString()` example above. This sets the string variable `args` equal to the value of `--log_level=test_suite` as specified in the first line of the `YAML` file.

If you trace the flow of the `test_radiosonde_opr` executable, you’ll soon come to the heart of the test suite, which is defined in `oops/src/test/interface/ObsOperator.h`. To understand the full structure of this file we refer you to our page on *JEDI Testing*. For our purposes here, we will pick up the action in the `test::testSimulateObs()` function template, which is one of the tests called by `test_ufo_radiosonde_opr`:
template <typename MODEL> void testSimulateObs() {

    [...] 
    const eckit::LocalConfiguration obsconf(TestEnvironment::config(), "Observations ");
    std::vector<eckit::LocalConfiguration> conf;
    obsconf.get("ObsTypes", conf);

This illustrates an important point, namely that new configuration objects are constructed through the derived (child) class of eckit::LocalConfiguration rather than the base class of eckit::Configuration (whose constructors are protected). The constructor shown here takes two arguments. The first is the output of the TestEnvironment::config() method. This returns a copy of the Configuration object that was used to create and initialize the test::TestEnvironment object itself, as shown above. The second argument is a string that serves to extract a component of that Configuration, in particular, everything contained under the Observations section of the YAML file. This component is placed in the LocalConfiguration object obsconf.

YAML and JSON objects are hierarchical and self-similar. So, the Observations component of the YAML file can be treated as a self-contained YAML object in its own right, with its own components. Configuration objects are the same way. One can define an eckit::Configuration object that includes the contents of the entire YAML file, as is the case for TestEnvironment::config(), or one can define an eckit::Configuration object that contains only a particular component of the top-level YAML structure, as is the case for obsconf. Remember that LocalConfiguration objects are Configuration objects since the former is a child (derived class) of the latter.

It's tempting to think of LocalConfiguration objects as components of Configuration objects but this is incorrect. One could in principle have an eckit::LocalConfiguration object refer to the YAML file as a whole and an eckit::Configuration object refer to a single section, though this is rarely done. The Local in LocalConfiguration refers to a local component of the JEDI code, not a local section of the YAML file. You can create, access, and even change eckit::LocalConfiguration objects in a way that is not possible with eckit::Configuration objects. In short, LocalConfiguration objects are local instantiations of Configuration objects that you can use to access the configuration file.

Variables, parameters, and other settings in the config file can be read by means of the various get() methods of the eckit::Configuration class. Paths are relative to the top-level of the YAML/JSON hierarchy that is contained in the Configuration object. Two examples are shown above. Since the TestEnvironment::config() object contains the entire YAML file, the top level of the hierarchy includes the top-level components of the YAML file, namely the variables test_framework_runtime_config, window_begin, and window_end, as well as the multi-component YAML objects LinearObsOpTest and Observations. The first of these top-level variables is read using the config.getString() method and placed into the local variable args. One could access other levels of the hierarchy using periods as separators, for example:

    std::cout << "The TL tolerance is: " << TestEnvironment::config().getDouble("linearObsOpTest.toleranceTL") << std::endl;

In the second example shown above, the obsconf object only contains the Observations section of the YAML file. At the top level of this section is ObsTypes, which is itself a vector of configuration objects. Our example YAML file only includes one item in ObsTypes, namely Radiosonde, but other Applications may include more. Since ObsTypes can include multiple components, the ObsType: Radiosonde declaration in the YAML file is preceded by a dash: - ObsType: Radiosonde (recall that this indicates a sequence or list in YAML). So, in order to read this component of the YAML file, the second code segment above first defines the variable conf as a vector of LocalConfiguration objects. Then it uses the eckit::Configuration::get() method to read it from the YAML file.

Note another feature of the Configuration class highlighted in the two examples above. One uses a specific getString() method to retrieve a string, the other uses a generic get() interface to retrieve a vector of LocalConfiguration objects. Both options are available. For further details see the Summary of Configuration Methods below.

1.5. Building, Testing, and Running JEDI
The `eckit::Configuration` class also has a few more methods that are extremely useful for querying the configuration file. The first is `has()`, which accepts one string argument (`std::string`) and returns a Boolean `true` or `false` depending on whether or not an item of that name exists in the Configuration file (at the level represented by the Configuration object). The second is `keys()`, which returns the items at a particular level of the YAML/JSON heirarchy.

As an example of how to use these query functions, we could place the following code after the code segment above from the `testSimulateObs()` function:

```cpp
std::string obstype = conf[0].getString("ObsType");
std::cout << obstype << " Keys: " << conf[0].keys() << std::endl;
if(conf[0].has("variables") ) {
    std::vector<std::string> vars = conf[0].getStringVector("variables");
    std::cout << obstype << " Variables " << vars << std::endl;
} else {
    std::cout << obstype << " Warning: Observations variables not specified in config file " << std::endl;
}
if(conf[0].has("Output")) {
    const eckit::LocalConfiguration outconf(conf[0], "Output");
    std::string outfile = outconf.getString("filename");
    std::cout << obstype << " Output file: " << outfile << std::endl;
} else {
    std::cout << obstype << " Warning: Observations Output not specified in config file " << std::endl;
}
```

Given the YAML file above, the output of this would be:

```
Radiosonde Keys: [GeoVaLs,ObsBias,ObsData,ObsFilters,ObsType,rmsequiv,tolerance,variables]
Radiosonde Variables: [air_temperature,eastward_wind,northward_wind]
Radiosonde Warning: Observations Output not specified in config file
```

This example illustrates again the stylistic principle noted above: YAML/JSON keys that represent single variables or vectors are rendered in lower case while those that represent configuration objects in their own right are rendered in CamelCase.

**Summary of C++ Configuration Methods**

In this section we summarize some of the most useful public methods available in the `eckit::Configuration` class and, by extension, the `eckit::LocalConfiguration` class.

Available methods for querying the configuration file include:

```cpp
virtual bool has(const std::string &name) const;
std::vector<std::string> keys() const;
```

Available methods for reading specific data types include:

```cpp
bool getBool(const std::string &name) const;
int getInt(const std::string &name) const;
long getLong(const std::string &name) const;
std::size_t getUnsigned(const std::string &name) const;
std::int32_t getInt32(const std::string &name) const;
std::int64_t getInt64(const std::string &name) const;
float getFloat(const std::string &name) const;
```
double getDouble(const std::string &name) const;
std::string getString(const std::string &name) const;
std::vector<int> getIntVector(const std::string &name) const;
std::vector<long> getLongVector(const std::string &name) const;
std::vector<size_t> getUnsignedVector(const std::string &name) const;
std::vector<int32_t> getInt32Vector(const std::string &name) const;
std::vector<int64_t> getInt64Vector(const std::string &name) const;
std::vector<float> getFloatVector(const std::string &name) const;
std::vector<double> getDoubleVector(const std::string &name) const;
std::vector<std::string> getStringVector(const std::string &name) const;
LocalConfiguration getSubConfiguration(const std::string &name) const;
std::vector<LocalConfiguration> getSubConfigurations(const std::string &name) const;

Each of these methods also has a version that accepts a second argument (of the same type as the return value) that will be used as a default value in the event that the item in question is not found in the configuration file.

Available generic interfaces for the get() method include:

virtual bool get(const std::string &name, std::string &value) const;
virtual bool get(const std::string &name, bool &value) const;
virtual bool get(const std::string &name, int &value) const;
virtual bool get(const std::string &name, long &value) const;
virtual bool get(const std::string &name, long long &value) const;
virtual bool get(const std::string &name, size_t &value) const;
virtual bool get(const std::string &name, float &value) const;
virtual bool get(const std::string &name, double &value) const;
virtual bool get(const std::string &name, std::vector<int> &value) const;
virtual bool get(const std::string &name, std::vector<long> &value) const;
virtual bool get(const std::string &name, std::vector<long long> &value) const;
virtual bool get(const std::string &name, std::vector<size_t> &value) const;
virtual bool get(const std::string &name, std::vector<float> &value) const;
virtual bool get(const std::string &name, std::vector<double> &value) const;
virtual bool get(const std::string &name, std::vector<LocalConfiguration> &value) const;

The Boolean return value reflects whether or not these items are found in the config file.

**Fortran Usage**

ECMWF also offers a Fortran interface to eckit called fckit that provides Fortran interfaces to many of the eckit::Configuration methods described in our Summary of Configuration Methods above.

However, JEDI does not currently use these fckit interfaces for accessing config files. Instead, JEDI defines its own Fortran interfaces to C++ oops::Configuration objects. These are defined in the file oops/src/oops/util/config_mod.F90 and they currently include the following Fortran subroutines:

```fortran
logical function config_element_exists(c_dom,query)
integer function config_get_int(c_dom,query,idefault)
real(kind=kind_real) function config_get_real(c_dom,query,rdefault)
function config_get_string(c_dom,length,query,sdefault)
function config_get_string_vector(c_dom, length, query)
```

The first argument in each of these routines (c_dom) is a pointer to the eckit::Configuration object in C++ that provides access to the config file as described above. These, like other interfaces in JEDI, use the intrinsic
ISO_C_BINDING Fortran module to pass information between C++ and Fortran. Within this framework, c_dom is declared as a pointer of type c_ptr, with an intent(in) attribute.

The query argument in the subroutines above is the name of the variable one wishes to retrieve from the config file (rendered as type character(*)). The config_get_real(), config_get_int(), and config_get_string() routines also include an optional default value to be used if the variable in question is not found in the config file. The two string functions also require the user to specify the length of the string to retrieve, which is passed as an integer length argument. In the case of the string vector, this refers to the length (number of characters) of each element of the vector; the number of elements is determined automatically by querying the config file.

As an example of how these Fortran interfaces are used, we’ll consider a code segment from the atmprofile_setup_() routine in the file ufo/src/ufo/atmosphere/atmprofile/ufo_atmprofile_mod.F90. This routine is called during the execution of the test_ufo_radiosonde_opr test that we have been considering throughout this document. It’s function is to set up the Fortran counterpart of the C++ ufo::ObsAtmProfile object that contains the Radiosonde observation operator.

```fortran
subroutine atmprofile_setup_(self, c_conf)
  use config_mod
  implicit none
  class(ufo_atmprofile), intent(inout) :: self
  type(c_ptr), intent(in) :: c_conf

  integer :: ii

  !> Size of variables
  self%nvars = size(config_get_string_vector(c_conf, max_string, "variables"))
  !> Allocate varout: variables in the observation vector
  allocate(self%varout(self%nvars))
  !> Read variable list and store in varout
  self%varout = config_get_string_vector(c_conf, max_string, "variables")

  [...]```

The first thing to note is that this routine uses the config_mod module in oops, which contains the configuration interface, as described above. One must also use iso_c_binding, which defines c_ptr and other data types (in this example, this declaration is done at the ufo_atmprofile_mod module level).

The setup routine then calls config_get_string_vector() twice; once to determine the number of variables listed in the config file and a second time to actually read the data. The first call is used to allocate the Fortran string vector that will contain the data. The length of each string buffer is set equal to the parameter max_string, which is also defined in the config_mod module.

Note that the various config_get*() routines retrieve data relative to the the top level of the eckit::Configuration object referred to by c_conf. As discussed above, each section of the YAML or JSON file can be rendered as self-contained eckit::Configuration object and the appropriate section of the config file is generally extracted in C++ and passed to the Fortran routines. In this example, the c_conf pointer points to the ObsType: Radiosonde section of the YAML file, as defined by the conf[0] object in the testSimulateObs() code segment above.

We could add the following code segment to the subroutine above to illustrate a few other features of the Fortran configuration interface:

```fortran
if (config_element_exists(c_conf,"GeoVaLs")) then
  write(*,*) "Radiosonde GeoVaLs Norm = ", config_get_real(c_conf,"GeoVaLs.norm",1.0_˓→kind_real)
endif```

78 Chapter 1. Table of Contents
Here we see that `config_element_exists()` is an interface to the `eckit::Configuration::has()` method discussed above that returns a Boolean `true` or `false` and that can be used to check if a variable exists in the config file. Furthermore, the period acts as a separator that can be used to access any level of the YAML/JSON hierarchy that is at or below the level defined by `c_conf`. Here we use it to access the `norm` element of the `Observations.ObsTypes[0].GeoVaLs.norm` item of the original YAML file. We also included a default value of unity to be used if the `config_get_real()` routine failed to find this variable in the config file. But, in our example, the variable exists and the output is:

```
Radiosonde GeoVaLs Norm = 8471.8836878543570
```

1.5.5 JEDI Configuration Files: Content

This document describes some of the items included in JEDI configuration (config) files and how to use them. It is not intended to be complete. Different models and observation types have different implementations of these basic elements, often with different parameters and variable names.

Users and developers are welcome to add their own items and the accompanying code to process them. We recommend using the JEDI repositories `ioda`, `ufo`, and `fv3-jedi` as prototypes for the structure, content, and naming conventions in the config files.

This document is intended only as a guide to help users interpret and customize JEDI configuration files in order to run applications. Developers may also use this information to write new tests. For practical information on how to modify the JEDI code to read and process the information contained in the configuration files see: JEDI Configuration Files: Implementation.

The sections of this document refer to the top-level items in the YAML/JSON configuration files and how each of these top-level items are used. This is a work in progress. There are plans to standardize the top-level contents of the JEDI config files and this document will be updated as this occurs.

**resolution/Geometry**

JEDI config files use both `resolution` and `Geometry` to define the model grid (both horizontal and vertical) and its parallelization across compute nodes. The latter term, `Geometry` is commonly used for unit tests at the oops level while `resolution` is often used for applications such as 4DEnsVar where multiple grids may be used. There are plans to standardize this terminology in the future but both terms are currently used.

Sometimes, as is the case with FV3, grid information is read from data files that are provided with the model repository via `git LFS`. This entry may also include one or more Fortran namelist files that are read by the model to set up the grid and its parallel partitioning.

**State**

Used to define multiple unit tests for the model state, including file IO, interpolation, variable changes, increments, and computation of the background error covariance matrix.

**Model, LinearModel**

Used to define model parameters, control flags, physics packages, and other options for the model and the linearized model respectively. Items may also define Fortran namelist files to be used by the model.

1.5. Building, Testing, and Running JEDI


**Covariance**

Provides information and specifications for computing the background error covariance matrix, also known as the B matrix. The first item in this section is often the variable **covariance**, which identifies the method by which the B matrix is computed. A popular option here is **BUMP**, the Background error covariance on an Unstructured Mesh Package developed by Benjamin Menetrier (JCSDA/Meteo-France) and distributed as part of oops. Alternatively, individual models may also offer a static B matrix.

**ModelBias, ModelBiasCovariance**

Used to define the model bias and its associated error covariance matrix.

**Initial Condition**

This is used to define the initial condition of a forecast or DA cycle. It often includes references to restart files and Fortran namelists to be used by the model upon startup. Alternatively, it may specify one of several idealized analytic states that can be used to initialize models. JEDI currently provides several options for analytic initialization based on the idealized benchmarks defined by the multi-institutional 2012 Dynamical Core Intercomparison Project sponsored by NOAA, NSF, DOE, NCAR, and the University of Michigan. Other analytic models may be added in the future.

**Observations**

Often the largest section of the configuration file, this describes one or more observation types (**ObsTypes**), each of which is a multi-level YAML/JSON object in its own right. As described our **Configuration file implementation example**, each of these **ObsTypes** are read into JEDI as an *eckit::Configuration* object. Commonly used components within each ObsType include

- **ObsType**: A name describing the observation type
- **ObsData**: Input and output file names that contain the data
- **variables**: Identifies variables required for the forward operator
- **Covariance**: Provides information and specifications for computing the observation error covariance matrix.
- **ObsFilters**: Used to define QC filters
- **ObsBias**: Used to specify the Bias correction
- **GeoVaLs**: Identifies simulated ufo output files and other parameters that are used for testing

Here is an **example YAML file** showing how to specify the creation of an output file from IODA.

**Assimilation Window**

Used to define the assimilation window for many applications, such as MakeObs, HofX, and EnsHofX.

**cost_function**

Specifies parameters, variables, and control flags used to define how the cost function should be calculated.
minimizer

This tells oops which algorithm to use for minimizing the cost function, specified by the variable `algorithm`. Valid options include DRGMRESR, DRIPCG, GMRESR, IPCG, SaddlePoint, RPCG, DRPCG, DRPFOM, LBGMRESR, DRPLanczos, PCG, PLanczos, RPLanczos, MINRES, and FGMRES.

Output

Used to specify the name, path, format, frequency, and other attributes of any output files that the application may produce.

Top-Level Variables

Most of the content in the JEDI config files is contained in sections of the YAML/JSON hierarchy that can themselves be treated as self-contained Configuration objects. Some of the more commonly used sections are described above, throughout this document. However, occasionally you will also find variables specified in the top level of the YAML/JSON hierarchy that are not part of a distinct section. These are often concerned with high-level operations such as defining the test suite, parallel configuration, IO frequency and log verbosity. In some cases, such as the ufo unit tests, this may also include high-level data assimilation parameters such as `window_begin` and `window_end`. These are read in as `util::DateTime` objects and are used to defined the assimilation window used for the tests.

1.6 JEDI Components

1.6.1 IODA

IODA Introduction

Overview

The Interface for Observation Data Access (IODA) component of JEDI provides the interfaces that bridge the external observation data to the components within JEDI that utilize those data, namely OOPS and UFO. As shown in Fig. 1.1, data flow in two directions through the IODA subsystem.

Data stored in observation data “tanks” at different centers are extracted, loaded into memory and presented through an interface to the JEDI OOPS and UFO subsystems. The DA run produces various results such as H(x), O-A, and O-B, which are (selectively) written out into files that can be used for plotting and other diagnostic purposes.

The goal for IODA is to be able to handle an immense amount of data from the tanks (Fig. 1.1, left side) without overwhelming the DA run. In addition, IODA needs to present observation data in different manners (e.g., spatially localized clusters of observations) for consumption by UFO and OOPS as well as write DA results in a format amenable to the user’s diagnostics tools.

Representation of Model and Observation Data

Earth system quantities (variables such at temperature, humidity, ocean salinity) are typically represented as 4-dimensional fields. Three spatial dimensions and time (x,y,z,t) comprise the total dimensions of these fields. For the spatial dimensions, a common way to define x and y (horizontal) are longitude and latitude, respectively, however the vertical dimension can be represented in many ways such as height, pressure, and a variety of terrain-following systems. Using atmospheric temperature (T) as an example, the variable is mathematically represented as T(x,y,z,t) in the governing equations. Model fields are commonly dense arrays with every element defined over the model domain,
Fig. 1.1: High level data flow through the IODA subsystem
so these fields are typically stored in 4D arrays. Observations on the other hand tend to be sparse (relative to the model representation), so these can be thought of as 2D arrays that are dimensioned by variables versus locations. Each location is an (x,y,z,t) tuple that defines where the corresponding observation value is located in the 4D field.

Note: Some observation quantities have been derived from integrations that effectively eliminate the vertical dimension (for example, satellite observations of radiance, brightness temperature, GNSSRO, etc.) These observations typically have horizontal spatial and temporal coordinates (x,y,t) associated with each observation value. In these cases, the location is an (x,y,t) tuple instead where the z component is not present.

Because of the sparse nature of the observation data, these data are viewed within JEDI as 2D arrays (variables versus locations) and the associated location values (the x, y, z, t coordinate values) are treated as meta data. Location #1 is a particular combination of x, y, z, t values, location #2 is another combination of x, y, z, t values, and so forth. During the execution of the DA run, the observation data and location meta data are passed around in objects to the subsystems that require these data.

JEDI Components that are Clients of IODA

As seen in Fig. 1.1, the clients of IODA within the JEDI system are UFO and OOPS. UFO is responsible for computing the simulated observations, H(x), from the model fields. Therefore, UFO takes the observation location meta data from IODA, queries the model for the field values at those locations and then runs the corresponding forward operators to calculate H(x) at all of the observation locations. One task for OOPS is to run minimization which operates in part on the difference between the actual observations, y, and the simulated observations, H(x). OOPS collects H(x) from UFO and y from IODA to form a departure, y - H(x), that is used in the minimization process.

External Observation Data

The data centers (eg. NCEP, in Fig. 1.1) store observation data in a wide variety of formats. Because of this, the general idea in IODA is to convert these formats to a common format, the “IODA data store”, to facilitate access by IODA. This way, the different manners in which JEDI may want to select, query, or distribute the observation data can be done through one API to the IODA data store format, as opposed to many API’s to the various data center formats. After the DA run completes, it is often desired to analyze different aspects of that run. IODA will provide a diagnostics file for this purpose whose contents will be selectable via the DA run configuration. “O minus A” is an example of a quantity that could be saved in the diagnostics file.

IODA Class Structure

Relationship Between OOPS and IODA

OOPS provides an abstract interface layer for classes involved in the DA operation. The OOPS classes are templated so that a variety of concrete objects can be instantiated for those classes. For example, in the current JEDI system there are two “toy” models, lorenz95 and qg, that implement their own concrete ObsVector class. In addition, IODA provides a third concrete ObsVector class. The selection of the ObsVector for a particular DA run is accomplished by declaring which ObsVector you want through the <MODEL> template in the OOPS abstract ObsVector class.

IODA provides concrete implementations for two of the OOPS abstract classes, ObsVector and ObsSpace, as shown in Fig. 1.2.

The transition from the OOPS abstract class to the corresponding IODA concrete class is handled through a pointer to an instantiated object of the IODA class. In the ObsVector case, the selection of the IODA ObsVector class is done by substituting “ioda::ObsVector” for the <MODEL> type in the OOPS template, and the OOPS data member “data_” is set up to point to an object of the IODA ObsVector.
Fig. 1.2: Class relationships between OOPS and IODA
The OOPS classes also define the interface to the concrete objects that lie below. The member methods of the OOPS class simply call the same named methods, in the instantiated object, with the same list of arguments for those methods in the instantiated object.

Multiple ObsSpace Objects

Note that the ObsSpace structure contains a set of ObsSpace objects (Fig. 1.2). At the top of the structure in OOPS, there is a class, ObsSpaces (note the plural), that contains a vector of pointers (data member “spaces”) that reference a set of OOPS ObservationSpace objects with their corresponding IODA ObsSpace objects.

Each IODA ObsSpace object instantiated by OOPS ObsSpaces is associated with a corresponding UFO ObsOperator. The ObsOperator is the object that simulates observation values, H(x). The ObsSpace, ObsOperator pairs form according to observation type (e.g., radiance, radiosonde, aircraft) that only work on those types as opposed to the entire set of observations. This breaks the problem up into smaller more manageable pieces, and facilitates the observation simulation since each observation type tends to require a different algorithm for its simulation.

Note that there will exist multiple ObsVector objects for two reasons. First because of the multiple ObsSpace objects, and second because both actual observations (the y vector) and simulated observations (the H(x) vector) are stored in ObsVectors. Each ObsVector is associated with a single ObsSpace that stores its values in the corresponding Obs Database (Fig. 1.2).

IODA Data Flow

The primary use of IODA in the JEDI system is to manage the actual observations in the y vector, and simulated observations in the H(x) vector. Both of these quantities are observation vectors which are stored in ObsVector objects. Fig. 1.3 depicts the manner in which y and H(x) are presented to JEDI for use in the DA cost function.

Two OOPS Observations objects are created to hold the total y and H(x) vectors, and one Observer object is created to transform the x vector (from the model) into the H(x) vector. The Observations and Observer objects are how the rest of JEDI access the y and H(x) vectors. Note that the resulting H(x) from the set of ObsOperator objects is stored in an Observations object since H(x) is a type of observation vector.

At various points in the DA run, results such as H(x), O-A, or O-B (i.e., any quantity corresponding to a observation vector) can be stored in an Observations object for subsequent writing out into a results file. An example for H(x) is shown in Fig. 1.3 with the orange arrow pointing from the set of ObsSpace objects to the Obs Data file. The data written into the results file can later be used for analysis of the DA run.

IODA Interfaces

Background

IODA interacts with external observation data on one side and with the OOPS and UFO components of JEDI on the other side (Fig. 1.1). On the observation data side there exist huge amounts of data, thus creating the need to pare these data down to a manageable size for JEDI during a particular DA run. The primary mechanism for accomplishing this is to filter out all of the observations that lie outside the current DA timing window, and present only that subset to be read into memory during a DA run.

There are many different types of observations that come with a variety of ways that the observation data are organized. To the extent that is feasible, it is desirable to devise a common data organization of which all of these observation types can employ. The memory representation of observation data in IODA has started with a prototype along these lines that successfully places a number of observation types (radiosonde, aircraft, AMSU-A, GNSSRO, plus several more) into a common organization.

At this point, we have a prototype architecture defined for the handling of files containing observation data (Fig. 1.4).
IODA Data Flow

Fig. 1.3: Data flow through IODA, UFO and OOPS objects
Fig. 1.4: IODA file handling
The intent of this architecture is to enable the use of one IODA file reader/writer implementation, namely the IodaIO class in Fig. 1.4. Using the single IodaIO class will make future maintenance much simpler, especially since we have not yet settled on the particular file format to use for the IODA Datafile piece.

A first pass implementation of this architecture has been created in the ioda-converters github repository. This implementation is not quite in the form of the prototype architecture, but is evolving toward that goal. Currently, we are using netcdf for the IODA Datafile format (subject to change) and we have a common netcdf writer in the ioda-converters with a collection of readers for the various observation data file formats. Work is in progress to evolve the current implementation to the prototype architecture.

A prototype interface, using a common data organization (Fig. 1.5), has been defined for access to observation data from the JEDI components OOPS and UFO.

Central to this scheme are the 2D arrays holding the observation data quantities (ObsValue, ObsError, HofX in Fig. 1.5). Each row of the ObsData array holds a particular vector of observation-related data. The length of each row is equal to the number of unique locations (nlocs), (x,y,z,t) or (x,y,t) tuple values, and the number of rows corresponds to the set of available observation variables (nvars). Note that an ObsVector can be constructed using a subset of the variables (rows) in the ObsData arrays. In the case of satellite data, the rows correspond to individual instrument channels.

Currently, the elements in an ObsData array are either floating point numbers or integers (e.g., QC marks). As more complex observation types come on board, the idea is to enhance the ObsData array so that each element can be a more complex data type. For example, each element could be a vector, or a higher rank array of numbers. In the case of using vectors for each element, the ObsData array effectively becomes a three-dimensional array.

In addition to the ObsData arrays, two more arrays are added that contain meta data. The Location Meta Data array (Fig. 1.5) contains rows, of length nlocs, corresponding to meta data oriented by location. Examples of Location Meta Data are quantities that describe each location such as Latitude, Longitude, Date/Time, and descriptive quantities.
associated with locations such as the Scan Angle of a satellite-borne instrument.

The Variable Meta Data array is analogous to the Location Meta Data array, except that it holds meta data associated with the variables in the ObsData arrays. Examples include the variable names, and in the case of some instruments, channel numbers and channel frequencies.

A first pass implementation of this interface has been implemented in the ioda github repository. This implementation is entirely in C++ and is successfully handling a small set of observation types including radiosonde, aircraft, ADO, AMSU-A, GNSSRO and Marine (SST, sea ice thickness and fraction, etc.) test cases.

**Interfaces to External Observation Data**

These interfaces are under heavy development and currently not well defined. We are working with data providers to get these interfaces more clearly defined over the next one to two months.

**Data Tanks**

The means for converting observation data in the external data tanks into files that IODA can read are being handled by a number of scripts and programs in the ioda-converters github repository. This code is relatively new and under active development. The goal is to organize the code into specific readers for each data tank format, all tied into a general IODA file writer, namely the IodaIO abstract interface class shown in Fig. 1.4. Organizing this way will allow us to experiment with different file formats, for the IODA datafile piece (Fig. 1.4), with minimal interference for the clients of the IodaIO class.

**Diagnostic Files**

At this point, we are actively investigating the best option for the diagnostic file type and data organization in IODA. We are using the same data organization as the IODA input file which currently is a netcdf file. As the requirements for downstream diagnostic tools get clarified, the file type and data organization are subject to change.

The creation of the diagnostics file (IODA output) is specified in the YAML configuration. The ObsDataOut keyword along with the obsfile sub-keyword are used to request that a diagnostics file be created. This occurs during the destructor of the ObsSpace object, which is near the end of the DA run. Currently, the entire contents of the memory store is written into the output file, and there are plans to allow for the selection of a subset of the memory store via the YAML configuration.

If the DA run is using multiple process elements, one file per element is created using just the observation data associated with that element. The file names get the process element rank number appended to them which avoids file collisions. This scheme is okay for testing with small datasets, but could be problematic when using a large number of process elements. This will need to be addressed before getting into operational sized DA runs.

**Example Radiosonde YAML**

The following is the YAML for the UFO test “test_ufo_radiosonde_opr”.

```
---

test_framework_runtime_config: "--log_level=test_suite"
window_begin: '2018-04-14T21:00:00Z'
window_end: '2018-04-15T03:00:00Z'
LinearObsOpTest:
  testiterTL: 12
  toleranceTL: 1.0e-9
```

(continues on next page)
toleranceAD: 1.0e-11

Observations:
- ObsTypes:
  - ObsType: Radiosonde
    ObsData:
      ObsDataIn:
        obsfile: Data/sondes_obs_2018041500_m.nc4
      ObsDataOut:
        obsfile: Data/sondes_obs_2018041500_m_out.nc4
    variables:
      - air_temperature
    GeoVaLs:
      random: 0
      filename: Data/sondes_geoval_2018041500_m.nc4
      window_begin: '2018-04-14T21:00:00Z'
      window_end: '2018-04-15T03:00:00Z'
      vecequiv: GsiHofX
      tolerance: 1.0e-04  # in % so that corresponds to 10^-3
    ObsBias: {}

Under the ObsType: Radiosonde specification, the output file is requested to be created in the path: Data/sondes_obs_2018041500_m_out.nc4. If there is only one process element, then the output will appear in the file as specified. However, if there are 4 process elements, then the output will appear in the following four files:

- Data/sondes_obs_2018041500_m_out_0000.nc4
- Data/sondes_obs_2018041500_m_out_0001.nc4
- Data/sondes_obs_2018041500_m_out_0002.nc4
- Data/sondes_obs_2018041500_m_out_0003.nc4

More details about constructing and processing YAML configuration files can be found in JEDI Configuration Files: Content and JEDI Configuration Files: Implementation.

Interfaces to other JEDI Components

These interfaces have a much clearer definition than the interfaces to external observation data (see Interfaces to External Observation Data above). However, these are still new and will likely need to evolve as more observation types are added to the system.

OOPS Interface

OOPS accesses observation data via C++ methods belonging to the ObsVector class. The variables being assimilated are selected in the YAML configuration using the variables sub-keyword under the ObsType keyword. In the radiosonde example above, one variable “air_temperature” is being assimilated. In this case, the ObsVector will read only the air_temperature row from the ObsData table and load that into a vector.

The ObsVector class contains the following two methods, read() for filling a vector from an ObsData array in memory and save() for storing a vector into an ObsData array.

```c++
// Interface prototypes
void read(const std::string &
void save(const std::string & const;
```

- The std::string arguments are the names of the ObsData array that is to be accessed.
Following is an example of reading into an observation vector. Note that the ObsVector object \( yobs_ \) has already been constructed which included the allocation of the memory to store the observation data coming from the \texttt{read()} method.

```cpp
// Read observation values
Log::trace() << "CostJo::CostJo start" << std::endl;
yobs_.read("ObsValue");
Log::trace() << "CostJo::CostJo done" << std::endl;
```

Here is an example of saving the contents of an observation vector, \( H(x) \), into an ObsData array. The ObsVector object \( yobs \) is constructed in the first line, and the third line creates an ObsData array called “hofx” and stores the vector data into that ObsData array.

```cpp
// Save \( H(x) \)
boost::scoped_ptr<Observations_> yobs(pobs->release());
Log::test() << "H(x): " << *yobs << std::endl;
yobs->save("hofx");
```

### UFO Interface

UFO accesses observation data via Fortran functions and subroutines belonging to the ObsSpace class. ObsSpace is implemented in C++ and a Fortran interface layer is provided for UFO. The following three routines are used to access observation data, and unlike the ObsVector methods in the \texttt{OOPS Interface} above, access is available to ObsData arrays and all Meta Data arrays. Reasons to access ObsData arrays from UFO would be for debugging purposes or for storing results, such as \( H(x) \), for post analysis. Typically, only meta data are used in the actual \( H(x) \) calculations.

#### Interface prototypes

- \texttt{integer function obsspace_get_nlocs(obss)}
- \texttt{subroutine obsspace_get_db(obss, group, vname, vect)}
- \texttt{subroutine obsspace_put_db(obss, group, vname, vect)}

- **The \textit{obss} arguments are C pointers to ObsSpace objects.**
- **The **\textit{group}** arguments are names of the ObsData arrays holding the requested variable  
  - E.g., “HofX”, “MetaData”**
- **The **\textit{vname}** arguments are names of the requested variable (row)  
  - E.g., “air_temperature”, “Scan_Angle”**
- **The \textit{vect} argument is a Fortran array for holding the data values  
  - The client (caller) is responsible for allocating the memory for the \textit{vect} argument**

Following is an example from the CRTM radiance simulator, where meta data from the instrument are required for doing the simulation.

```fortran
! Get nlocs and allocate storage
nlocs = obsspace_get_nlocs(obss)
allocate(TmpVar(nlocs))

! Read in satellite meta data and transfer to geo structure
call obsspace_get_db(obss, "MetaData", "Sat_Zenith_Angle", TmpVar)
geo(:)%Sensor_Zenith_Angle = TmpVar(:)
call obsspace_get_db(obss, "MetaData", "Sol_Zenith_Angle", TmpVar)
geo(:)%Source_Zenith_Angle = TmpVar(:)
```

(continues on next page)
call obsspace_get_db(obss, "MetaData", "Sat_Azimuth_Angle", TmpVar)
geo(:)%Sensor_Azimuth_Angle = TmpVar(:)

call obsspace_get_db(obss, "MetaData", "Sol_Azimuth_Angle", TmpVar)
geo(:)%Source_Azimuth_Angle = TmpVar(:)

call obsspace_get_db(obss, "MetaData", "Scan_Position", TmpVar)
geo(:)%Ifov = TmpVar(:)

call obsspace_get_db(obss, "MetaData", "Scan_Angle", TmpVar) !The Sensor_Scan_Angle is optional
geo(:)%Sensor_Scan_Angle = TmpVar(:)

deallocate(TmpVar)

An example for storing the results of a QC background check is shown below. Note that the storage for “flags” has been allocated and “flags” has been filled with the background check results prior to this code.

write(buf,*)'UFO Background Check: ',ireject,trim(var),' rejected out of ',icount,' (',iloc,' total)'
call fckit_log%info(buf)

! Save the QC flag values
call obsspace_put_db(self%obsdb, self%qcname, var, flags)

1.6.2 UFO

Creating new Observation Operator in UFO

Existing Observation Operators

Before implementing a new observation operator, check if one of the observation operators already implemented in UFO is suitable:

1. Interface to CRTM for radiances and aerosol optical depth (ufo/src/ufo/crtm/, “CRTM” for radiances, “AOD” for aod in the config files)

2. Linear vertical interpolation in log pressure for the variables specified in the config file (ufo/src/ufo/atmvertinterp, “Radiosonde”, “Aircraft”, “Satwind”, more names could be added to ObsAtmVertInterp.cc).

3. Identity observation operator for 2D fields (takes the result of the horizontal interpolation from the geovals and returns it as H(x)) (ufo/src/ufo/identity, “Surface”, more names could be added to ObsIdentity.cc)

4. Several GNSSRO observation operators.

Creating files for a new Observation Operator

If your observation operator is different from the above, you may need to create a new observation operator. Typically, all the files for a new observation operator are in a new directory under ufo/src/ufo.
The new observation operator has to have a C++ interface, because all observation operators have to be accessed by a generic data assimilation layer written in C++ in oops. Most of the observation operators, however, are written in Fortran. The directory for the observation operator consists of the following files (example from atmvertinterp):

1. ObsAtmVertInterp.cc, ObsAtmVertInterp.h: C++ files defining the ObsOperator class. The methods (functions) there call Fortran subroutines.
3. ufo_atmvertinterp_mod.F90: Fortran module containing the code to run observation operator.

Most of the time you’d only need to modify the Fortran module (3), and the files from (1-2) can be generated automatically.

To generate the ObsOperator files, you can run the following script: `ufo/tools/new_obsop/create_obsop_fromexample.sh <ObsOperatorName> <directory>`

`<ObsOperatorName>` is an UpperCamelCase name you’d like your obs operator to go by. `<directory>` is a directory name in `ufo/src/ufo`. Examples for existing obsoperators: atmvertinterp, crtm, identity.

Example of calling `create_obsop_fromexample.sh`:

```bash
$> ./create_obsop_fromexample.sh MyOperator myoperator
```

After the directory with the new obsoperator is created, add it to `ufo/src/ufo/CMakeLists.txt`:

```cmake
add_subdirectory( identity )
add_subdirectory( myoperator )
list( APPEND ufo_src_files
      ${identity_src_files}
      ${myoperator_src_files}
)
```

and try to compile/build the code.

### Adding an Observation Operator test

After this skeleton code is generated, create a test for your new observation operator. Even if the test fails because of missing data or a mismatch between computed and provided values, the test will still call your operator and any print statements or other calls you perform within the Fortran subroutines will execute.

For observation operator test one needs a sample observation file and a corresponding geovals file. Observation file should be added to ioda repository in `ioda/test/testinput/atmosphere/`. Corresponding geovals file should be added to ufo repository in `ufo/test/testinput/atmosphere/`.

All observation operator tests in UFO use the OOPS ObsOperator test. To create a new one, add an entry to `ufo/test/CMakeLists.txt` similar to:

```cmake
ecbuild_add_test( TARGET test_ufo_myoperator_opr
                  SOURCES mains/TestObsOperator.cc
                  ARGS "testinput/myoperator.yaml"
                  LIBS ufo )
```

Other changes required in `ufo/test/CMakeLists.txt`:

Link the `config file` you will be using for the test:

```cmake
list( APPEND ufo_test_input
      testinput/myoperator.yaml
)
Link the observations and geovals files you will be using for the test:

```
list( APPEND ufo_test_data
    atmosphere/geoval_file_name.nc4
)
list (APPEND ioda_obs_test_data
    atmosphere/obs_file_name.nc4
)
```

To configure the test, create config file `ufo/test/testinput/myoperator.yaml` and fill appropriately. For examples see `ufo/test/testinput/amsua_crtm.yaml`, `ufo/test/testinput/radiosonde.yaml`.

**Adding substance to the new Observation Operator**

To implement the Observation Operator, one needs to:

- Specify input variable names (requested from the model) and output variable names (simulated by the observation operator) in `ufo_obsoperator_mod.F90`, subroutine `ufo_obsoperator_setup`. The input variable names need to be saved in `self%varin` (set `self%nvars_in` and allocate accordingly), the output variables in `self%varout` (set `self%nvars_out` and allocate accordingly). See examples in `ufo/src/ufo/atmvertinterp/ufo_atmvertinterp_mod.F90` and `ufo/src/ufo/crtm/ufo_radiancecrtm_mod.F90`. The variables can be hardcoded or controlled from the config file depending on your observation operator.

- Fill in `ufo_obsoperator_simobs` routine. This subroutine is for calculating $H(x)$. Inputs: `geovals` (horizontally interpolated to obs locations model fields for the variables specified in `self%varin` above), `obss` (observation space, can be used to request observation metadata). Output: `hofx` (obs vector to hold $H(x)$). Note that the vector was allocated before the call to `ufo_obsoperator_simobs`, and only needs to be filled in.

**Observation Operator test**

All observation operator tests in UFO use the OOPS ObsOperator test from `oops/src/test/interface/ObsOperator.h`.

There are two parts of this test:

1. testConstructor: tests that ObsOperator objects can be created and destroyed
2. testSimulateObs: tests observation operator calculation in the following way:
   - Creates observation operator, calls `ufo_obsoperator_setup`
   - Reads “GeoVaLs” (vertical profiles of relevant model variables, interpolated to observation lat-lon location) from the geovals file
   - Computes $H(x)$ by calling `ufo_obsoperator_simobs`
   - Reads benchmark $H(x)$ from the obs file (netcdf variable name defined by `vecequiv` entry in the config) and compares it to $H(x)$ computed above
   - Test passes if the norm(benchmark $H(x)$ - $H(x)$) < tolerance, with tolerance defined in the config by `tolerance`.

94 Chapter 1. Table of Contents
1.7 Background

The long term objective of the Joint Effort for Data assimilation Integration (JEDI) is to provide a unified data assimilation framework for research and operational use, for different components of the Earth system, and for different applications, with the objective of reducing or avoiding redundant work within the community and increasing efficiency of research and of the transition from development teams to operations.

See the following links for additional background information.

- High-level requirements for JEDI
- General methodology for JEDI

1.8 Working Practices

Nowadays, software development is a collaborative activity, between members of a team and across teams, locally or spread over different cities or countries. Developers working on common software might even never physically meet. Nevertheless, work needs to be coordinated efficiently to avoid wasted effort. Modern software engineering practices make this routine achievement in the software industry. The infrastructure proposed here in the context of weather forecasting and related developments relies on those modern software engineering practices. It enables fast and easy engagement, flexible code management and proper control of operational releases.

This section describes the working practices and governance for collaborative code development using the GitHub ecosystem (GitHub and ZenHub) and the git flow workflow. GitHub is a git management tool with online interface and repositories and ZenHub is a planning and issue tracking tool that links with git repositories. These tools are all cross connected to form an ecosystem that has become an industry standard. They provide the means for easy access and fast engagement while still allowing proper control at all levels.

This document is fairly general and most of it can be applied to any collaborative code development. Practical aspects for developers to start with the system are described separately.

See the following links for more details on the JEDI working practices.

- Branching and merging code
- Forking and cloning repositories
- Reviewing code
- Testing
- Creating documentation

1.9 Governance of a community system

The decision to include code or not into a community system depends on several criteria. Of course scientific and technical quality are among the criteria, but usefulness to the community is another very important one. It is in fact the most important. For every community project, there should be a governance body to make that decision, based on that sole criteria. The review process determines if the scientific and technical quality are sufficient at any point in time.

The decision not to include a certain aspect in the community code is not a judgment on its scientific excellence. It could be that aspects that are critical for one user are useful only for that user. In that case, the code should be kept in a separate repository and it is the build system that brings the codes together and includes what is required for a given application without affecting the others. In that respect, current efforts to modernise software architecture in the data assimilation and forecasting system are absolutely essential because previous programming technology did not make this possible.
In old style Fortran, separating code that was specific to a user from a community code meant that some subroutine calls would be left dangling and possibly some global variables would be left in an unknown state. This was addressed with dummy routines that would be provided with the common code. Unfortunately, this approach doesn’t scale and quickly becomes unmanageable. The solution was then to include everything in the shared code, which quickly became bloated, difficult to manage, and unpopular.

In modern programming this is common practice. On one hand, a specialised sub-class in an inheritance structure leaves no trace behind when it is removed. Examples can be the use of specific observation types in a DA system, or a specific physics package in a model and many other circumstances are possible. On the other side of the spectrum, a high level application constructed from a collection of objects does not have any impact on other applications using all or some of the same objects. Software packages might have only low level extensions (e.g. browsers or applications like photoshop that support plugins) or high level extensions (system libraries or MPI) or both (JEDI is in that category).

However, this will only work if the interfaces in the middle layer are stable. If interfaces of a base class change, all subclasses will need to adapt. If interfaces of the objects used by high level applications change all those applications will need to change. If that happens often, the system will quickly become unpopular. It is the second role of a governance body for a community unified data assimilation and forecasting system: ensure that changes in the interfaces are infrequent and fully justified, documented and communicated if they become necessary.

The last role of the governance body is to provide guidelines regarding who has authorization to review and administer code at each level, most importantly at the release preparation level. Typically, this means designating a small pool of reviewers for each main component of the code, and a criterion, such as a minimum number of reviewers approving the pull request, for accepting it. The size of the pool of reviewers and number of approvals should ensure enough scrutiny, while maintaining an efficient process. As explained above, reviewers should be trusted to add other reviewers or delegate their roles on a case by case basis, in particular for small changes.

### 1.10 Roles and Responsibilities

**Governance board:** The board comprises representatives from the organizations involved in the collaboration and the project lead(s). It makes high level decisions about the directions of development and designates the administrators of the central repository and senior reviewers.

**Project lead:** leads and coordinate code developments in the directions given by the board. Reports on progress and issues to the board.

**Administrators** (a.k.a. gatekeepers, project maintainers): are responsible for giving access at the repository or branch level to relevant collaborators. The administrators have the authority to merge pull requests after it has been approved by reviewers. Administrators can give advice and help developers when merging conflicts arise. Each forked repository should have at least one administrator.

**Reviewers:** check that a proposed pull request follows all the minimum requirements for the level at which it is to be merged, including coding standards, passing of relevant tests (that have been run by the developer) and scientific evaluation if applicable. In principle all developers should be involved in reviewing other developer’s code. Senior reviewers can be designated by the board to oversee reviews in particular areas of the code.

**Developers:** anybody who edits the code. Developers have the responsibility to document their developments, to update them to the level of the develop branch and to test them before submitting a pull request.

### 1.11 Conclusion

The code management structure described here provides mechanisms for a unified data assimilation and forecasting system that is open to the wider scientific community as well as tightly controlled for operational use.
The organisation structure provides clear roles for developers, reviewers and the governance body as well as guidelines for interactions between those roles, thus ensuring efficiency of the continuous development process.
CHAPTER 2

Indices and tables

- genindex
- search