Dead-link checker status (does not actually build guide itself): 

Guide

This is a guide to projects and software development at the Netherlands eScience Center. It both serves as a source of information for exactly how we work at the eScience Center, and as a basis for discussions and reaching consensus on this topic.

This Guide is a work in progress

Read this book online here: **https://guide.esciencecenter.nl**

If you would like to contribute to this book see **CONTRIBUTING.md**.

To see who is responsible for which part of the guide see **chapter_owners.md**.

The guide is best read online, but there is a **PDF document** for offline reading.

Software Development

In this chapter we give an overview of the best practices for software development at the Netherlands eScience Center, including a rationale.

Software checklist

Here we provide a short checklist for software projects, the rest of this chapter elaborates on the various point in this list.

The **bare minimum** that every software project should do, from the start, is:

- Pick & include an **open source license**
- Use **version control**
- Use a **publicly accessible** version controlled repository
- Add a **readme describing the project**

We recommend that you also do the following (from the start of the project):

- Use **code quality tools**
- **Testing**
- Use **standards**

Additional steps depend on the goal of the software (zero or more can apply):

- **I'm publishing a paper**
- **I'm expecting users**
- **I'm expecting contributors**

I'm publishing a paper

- Add a **CITATION.cff file**
- **Make your software citable**
- Cite DOI in paper

I'm expecting users

- **Release** your software
- Provide **user documentation**
- **Easy installation**
- Provide issue tracker

I'm expecting contributors

- Provide **development documentation**
- Provide a **means of communication**
- Implement and add a **code of conduct**
- **Contribution guideline**

Version control

Why would you use version control software and hosting (such as GitHub)?

- **Easier to collaborate** Version control makes it easier to work on the same code simultaneously, while everyone still has a well defined version of the software (in contrast to a google-docs or shared file system type of system). Moreover, version control hosting websites such as Github provide way to communicate in a more structed way, such as in code reviews, about commits and about issues.
- **Reproducibility** By using version control, you never lose previous versions of the software. This also gives you a log of changes and allows you to understand what happened.
- **Backup** Version control is usually pushed to an external a shared server, which immediately provides a backup.
- **Integration** Version control software and host makes it more easy to integrate with other software that support modern software development, such as testing (continuous integration ,automatically run tests, build documentation, check code style, integration with bug-tracker, code review infrastructure, comment on code).

GitHub

Netherlands eScience center uses GitHub **GitHub** for version control. To keep our code transparent and findable the preferred code hosting platform is GitHub and version

management is git. The repository should preferably be public from the start.

By default an eScience Research Engineer is expected to create a new **GitHub organization** for each project and create repositories in there. However a new repository should be made in the **Netherlands eScience Center GitHub organization (https://github.com/NLeSC)** when the repository is used in multiple projects.

Policy

- No repositories which the Netherlands eScience Center is paying for should be in personal accounts, they SHOULD always be in either the **Netherlands eScience Center GitHub organization** or in a project based GitHub organization
- GitHub supports **two-factor authentication**. This SHOULD be enabled for your account
- Project based GitHub organizations
  - MUST have at least two owners that are Netherlands eScience center employees
  - MUST be **registered** at **https://nlesc.github.io/**, to keep track of all the project organizations
  - Private repositories can be created. Free when **GitHub's education discount** is requested. **NOTE**: The **Netherlands eScience Center IP policy** applies to any software we contribute to, so the repository SHOULD become open source at some point. To prevent private repositories from remaining unnecessarily private forever please add a brief statement in the README of your repository, clarifying:
    - Why is this repository private?
    - On which date can this repository be made public?
    - Who should be consulted if we would like to make the repository public in the future?
- **Netherlands eScience center Github organization (https://github.com/NLeSC)**
  - Only Netherlands eScience center employees are members
  - All members have permission to create new repositories
  - **Collaborators** SHOULD be used to grant access to non-members
  - A limited number of slots for private repositories is available, but using them is discouraged
  - To prevent private repositories from remaining unnecessarily private forever please add a brief statement in the README of your repository, clarifying:
    - Why is this repository private?
    - On which date can this repository be made public?
    - Who should be consulted if we would like to make the repository public in the future?

Version control from the beginning of the project

It is highly recommended to start using version control on day one of the project.

Use git as version control system

Other version control systems can be used if the project does not start in the eScience Center and does not use git, or when the prevailing version control system in the particular community is not git. Even then, changing version control systems should be considered (especially if Subversion or another centralised system is used).

Git documentation:

- GitHub help: **http://help.github.com**
- Git homepage: **http://git-scm.com/**
- Pro Git Online Book: **http://git-scm.com/book**
- Reference: **http://gitref.org/index.html**
- In depth book: **Version Control with Git**
- for those who know subversion and want to learn git: **Git - SVN Crash Course**

Choose one branching model

A branching model describes how the project deals with different versions of the codebase, like releases and various development versions, and how to accept code contributions. Make the choice explicit in the contribution guidelines, and link to documentation on how to get started with it. Our default choice is **GitHub flow branching model**

GitHub flow is a very simple and sane branching model. It supports collaboration and is based on pull requests, therefore relies heavily on GitHub. The **Pro Git** book describes in detail the workflow of collaboration on the project with use of git branches, forks and GitHub in **Contributing to a Project chapter**. Other more complicated models could be used if necessary, but we should strive for simplicity and uniformity within the eScience Center since that will enhance collaboration between the engineers. Learning a new branching model should not stand in the way of contributions. You can learn more about those other models from **atlasian page**.

Repositories should be public

A public code repository has several benefits:

- It makes your code findable.
- It is a central point for users and collaborators.
- It shows your code to world, allowing (re)use and enables you to get credit for your work.
- It is usually not hosted on your laptop, and hence provides an external backup.

Unless code cannot be open (e.g. when working with commercial partners, or when there are competitiveness issues) it should be in a public online repository. In case the code uses data

that cannot be open, an engineer should try to keep sensitive parts outside of the main codebase. If you accidentally included copyrighted files in your repository, you need to remove them from the HEAD as well as from history. There is a **gist that explains how**.

Meaningful commit messages

Commit messages are the way for other developers to understand changes in the codebase. In case of using GitHub flow model, commit messages can be very short but pull request comments should explain all the changes. It is very important to explain the why behind implementation choices. To learn more about writing good commit messages, read **tpope's guide** and **this post**

GitHub has some interesting features that allow you to **close issues directly from commit messages**.

Code snippets library

Sometimes, we develop small snippets of code that can be directly reused in other projects, but that are too small to put in a library. We store these code snippets in git, in **GitHub Gists**.

Code Quality

Ways to improve code quality are in the **Code quality** chapter on the Turing Way.

There are **online software quality improvement tools** see the **language guides** for good options per language.

Editorconfig

The eScience Center has a **shared editor config file**

Name spaces

If your language supports namespaces, use **nl.esciencecenter** or better a namespace based on the project.

Code reviews

See the **Code Reviews** section.

Code reviews

Introduction

At the eScience Center, we value software quality. Higher quality software has fewer defects, better security, and better performance, which leads to happier users who can work more effectively.

Code reviews are an effective method for improving software quality. McConnell (2004) suggests that unit testing finds approximately 25% of defects, function testing 35%, integration testing 45%, and code review 55-60%. While that means that none of these methods are good enough on their own, and that they should be combined, clearly code review is an essential tool here.

Code review also improves the development process. By reviewing new additions for quality, less technical debt is accumulated, which helps long-term maintainability of the code. Reviewing lets developers learn from each other, and spreads knowledge of the code around the team. It is also a good means of getting new developers up to speed.

The main downside of code reviews is that they take time and effort. In particular, if someone from outside the project does the reviewing, they'll have to learn the code, which is a significant investment. Once up to speed, the burden is reduced significantly however, and the returns include a much smaller amount of time spent debugging later.

Approach

It's important to distinguish between semi-formal code *reviews* and formal code *inspections*. The latter involve "up to six participants and hours of meetings paging through detailed code printouts" (SMARTBEAR 2016). As this extra formality does not seem to yield better results, we limit ourselves to light-weight, informal code reviews.

Process

We haven't yet decided on how to integrate code reviews into our working process. While that gets hashed out, here is some general advice from various sources and experience.

- Review everything, nothing is too short or simple
- Try to have something else to do, and spread the load throughout your working day. Don't review full-time.
- Don't review for more than an hour at a time, after that the success rate drops quite quickly
- Don't review more than 400 lines of code (LOC) at a time, less than 200 LOC is better

- Take the time, read carefully, don't review more than 500 LOC / hour

Prerequisites

Before handing over a change or a set of code for review, the following items should be there for the reviewer to work with:

- Documentation on what was changed and why (feature, bug, issue #, etc.)
- Comments / annotations by the author on the code itself
- Test cases

Also, before doing a code review, make sure any *tools* have run that check the code automatically, e.g. checkers for coding conventions and static analysis tools, and the test suite. Ideally, these are run as part of the continuous integration infrastructure.

Review checklist

This section provides two checklists for code reviews, one for the whole program, and one for individual files or proposed changes.

In all cases, the goal is to use your brain and your programming experience to figure out how to make the code better. The lists are intended to be a source of inspiration and a description of what should be best practices in most circumstances. Some items on this list may not apply to your project or programming language, in which case they should be disregarded.

Excluded from this checklist

The following items are part of a software quality check, but are better done by an automated tool than by a human. As such, they've been excluded from this checklist. If tools are not available, they should be checked manually.

- Coding conventions (e.g. PEP 8)
- Test coverage

Rubric for assessing code quality

All code should be level 3 or 4.

Level	1	2	3	4

Level	1	2	3	4
names	names appear unreadable, meaningless or misleading	names accurately describe the intent of the code, but can be incomplete, lengthy, misspelled or inconsistent use of casing	names accurately describe the intent of the code, and are complete, distinctive, concise, correctly spelled and consistent use of casing	all names in the program use a consistent vocabulary
headers	headers are generally missing or descriptions are redundant or obsolete; use mixed languages or are misspelled	header comments are generally present; summarize the goal of parts of the program and how to use those; but may be somewhat inaccurate or incomplete	header comments are generally present; accurately summarize the role of parts of the program and how to use those; but may still be wordy	header comments are generally present; contain only essential explanations, information and references
comments	comments are generally missing, redundant or obsolete; use mixed languages or are misspelled	comments explain code and potential problems, but may be wordy	comments explain code and potential problems, are concise	comments are only present where strictly needed
layout	old commented out code is present or lines are generally too long to read	positioning of elements within source files is not optimized for readability	positioning of elements within source files is optimized for readability	positioning of elements is consistent between files and in line with platform conventions
formatting	formatting is missing or misleading	indentation, line breaks, spacing and brackets highlight the intended structure but erratically	indentation, line breaks, spacing and brackets consistently highlight the intended structure	formatting makes similar parts of code clearly identifiable

Level	1	2	3	4
flow	there is deep nesting; code performs more than one task per line; unreachable code is present	flow is complex or contains many exceptions or jumps; parts of code are duplicate	flow is simple and contains few exceptions or jumps; duplication is very limited	in the case of exceptions or jumps, the most common path through the code is clearly visible
idiom	control structures are customized in a misleading way	choice of control structures is inappropriate	choice of control structures is appropriate; reuse of library functionality may be limited	reuse of library functionality and generic data structures where possible
expressions	expressions are repeated or contain unnamed constants	expressions are complex or long; data types are inappropriate	expressions are simple; data types are appropriate	expressions are all essential for control flow
decomposition	most code is in one or a few big routines; variables are reused for different purposes	most routines are limited in length but mix tasks; routines share many variables instead of having parameters	routines perform a limited set of tasks divided into parts; use of shared variables is limited	routines perform a very limited set of tasks and the number of parameters and shared variables is limited
modularization	most code is in one or a few large modules; or modules are artificially separated	modules have mixed responsibilities, contain many variables or contain many routines	modules have clearly defined responsibilities, contain few variables and a somewhat limited amount of routines	modules are defined such that communication between them is limited

- no need to assess a level that is not relevant to the software
- level 2 implies that the features in level 1 are not present, level 4 implies that the features in level 3 are also present

This rubric is based on:

> **Stegeman, Barendsen, & Smetsers (2016). Designing a rubric for feedback on code quality in programming courses. In proceedings of the 16th Koli Calling International**

Conference on Computing Education Research. ACM.

Program level checklist

Here is a list of things to consider when looking at the program as a whole, rather than when looking at an individual file or change.

Documentation

Documentation is a prerequisite for using, developing and reviewing the program. Here are some things to check for.

- Is there a description of the purpose of the program or library?
- Are detailed requirements listed?
- Are requirements ranked according to MoSCoW?
- Is the use and function of third-party libraries documented?
- Is the structure/architecture of the program documented? (see below)
- Is there an installation manual?
- Is there a user manual?
- Is there documentation on how to contribute?
    - Including how to submit changes
    - Including how to document your changes

Architecture

These items are mainly important for larger programs, but may still be good to consider for small ones as well.

- Is the program split up into clearly separated modules?
- Are these modules as small as they can be?
- Is there a clear, hierarchical or layered, dependency structure between these modules?
    - If not, functionality should be rearranged, or perhaps heavily interdependent modules should be combined
- Can the design be simplified?

Security

If you're making software that is accessible to the outside world (e.g. a web application), then security becomes important. Security issues are defects, but not all defects are security issues. A security-conscious design can help mitigate the security impact of defects.

- Which modules deal with user input?
- Which modules generate output?
- Are input and output compartmentalised?

    - If not, consider making separate modules that manage all input and output, so validation can happen in one place
- In which modules is untrusted data present?
    - The fewer the better
- Is untrusted data compartmentalised?
    - Ideally, validate in the input module and pass only validated data to other parts

Legal

"I'm an engineer, not a lawyer!" is an oft-overheard phrase, but being an engineer doesn't give you permission to ignore the legal rights of the creators of the code you're using. Here are some things to check. When in doubt, ask your licensing person for advice.

- Are the licenses of all modules/libraries that are used documented?
- Are the requirements set by those licenses fulfilled?
    - Are the licenses included where needed?
    - Are copyright statements included in the code where needed?
    - Are copyright statements included in the documentation where needed?
- Are the licenses of all the parts compatible with each other?
- Is the project license compatible with all libraries?

File/Change level checklist

When you're checking individual changes (e.g. pull requests) or files, the code itself becomes the subject of scrutiny. Depending on the language, files may contain interfaces, classes or other type definitions, and functions. All these should be checked, as well as the file overall:

- Does this file contain a logical grouping of functionality?
- How big is it? Should it be split up?
- Is it easy to understand?
- Can any of the code be replaced by library functions?

Interfaces

- Is the interface documented?
- Does the concept it models make sense?
- Can it be split up further? (Interfaces should be as small as possible)

Note that most of the following items assume an object-oriented programming style, which may not be relevant to the code you're looking at.

Classes and types

- Is the class documented?

- Does it have a single responsibility? Can it be split?
- If it's designed to be extended, can it be?
- If it's not designed to be extended, is it protected against that? (e.g. final declarations)
- If it's derived from another class, can you substitute an object of this class for one of its parent class(es)?
- Is the class testable?
  - Are the dependencies clear and explicit?
  - Does it have a small number of dependencies?
  - Does it depend on interfaces, rather than on classes?

Function/Method declarations

- Are there comments that describe the intent of the function or method?
- Are input and output documented? Including units?
- Are pre- and postconditions documented?
- Are edge cases and unusual things commented?

Function/Method definitions

- Are edge cases and unusual things commented?
- Is there incomplete code?
- Could this function be split up (is it not too long)?
- Does it work? Perform intended function, logic correct, ...
- Is it easy to understand?
- Is there redundant or duplicate code? (DRY)
- Do loops have a set length and do they terminate correctly?
- Can debugging or logging code be removed?
- Can any of the code be replaced by library functions?

Security

- If you're using a library, do you check errors it returns?
- Are all data inputs checked?
- Are output values checked and encoded properly?
- Are invalid parameters handled correctly?

Tests

- Do unit tests actually test what they are supposed to?
- Is bounds checking being done?
- Is a test framework and/or library used?

Providing feedback

The main purpose of a code review is to find issues or defects in a piece of code. These issues then need to be communicated back to the developer who proposed the change, so that they can be fixed. Doing this badly can quickly spoil everyone's fun.

Perhaps the most important point in this guide therefore is that the goal of a code review is *not* to provide criticism of a piece of code, or even worse, the person who wrote it. *The goal is to help create an improved version.*

So, when providing feedback, stay positive and constructive. Suggest a better way if possible, rather than just commenting that the current solution is bad. Ideally, submit a patch rather than an issue ticket. And always keep in mind that you're not required to find anything, if the code is fine, it's fine. If it's more than fine, file a compliment!

Most of our projects are hosted on GitHub, so most results will be communicated through pull requests and issues there. However, if you find something particularly bad or weird, consider talking in person, where a lengthy, complicated, or politically sensitive explanation is easier to do.

Communicating results through GitHub

If you are reviewing a pull request on Github, comments should be added in the `Files changed` section, so they can be attached to a particular line of code. Make many small comments this way, rather than a big ball of text with everything in it, so that different issues can be kept separate. Where relevant, refer to existing Issues and documentation.

If you're reviewing existing code rather than changes, it is still handy to use pull requests. If you find an issue that has an obvious fix, you can submit a pull request with a patch in the usual way.

If you don't have a fix, you can add an empty comment to the relevant line, and create a pull request from that as a patch. The relevant line(s) will then light up in the pull request's `Files changed` overview, and you can add your comments there. In this case, either the pull request is never merged (but the comments processed some other way, or not at all), or the extra comments are reverted and replaced by an agreed-upon fix.

In all cases, file many small pull requests, not one big one, as GitHub's support for code reviews is rather limited. Putting too many issues into a single pull request quickly becomes unwieldy.

References

Atwood, Jeff (2006) **Code Reviews: Just Do It**

Burke, Kevin (2011) **Why code review beats testing: evidence from decades of programming research.**

McConnell, Steve (2004) Code Complete: A Practical Handbook of Software Construction, Second Edition. Microsoft Press. ISBN-13: 978-0735619678

SMARTBEAR (2016) **Best practices for code review.**

Licensing

Without a license, all **rights** are at the author of the code or data, and that means nobody else can use, copy, distribute, or modify it work without consent. A license gives this consent.

Software licences

Software licenses are explained in **The Turing Way** chapter.

Apache 2 license

Apache 2 license

The Apache License version 2.0 is the default choice for licensing software developed at the Netherlands eScience Center. Other licenses can be used in special cases, e.g. when we add to existing software that already has a different license (see below), or if there are commercial partners that require different licensing.

The formal text of the licence is here: **http://www.apache.org/licenses/LICENSE-2.0.html** An informal explanation of what that means is here: **http://www.oss-watch.ac.uk/resources/apache2**

License grant

Each source file in your program or library should start with the following copyright statement in a comment block at the top (but underneath a shebang line if present, for technical reasons):

```
Copyright <years> Netherlands eScience Center and <Legal entities of
Licensed under the Apache License, version 2.0. See LICENSE for detai
```

The same notice should be somewhere in your README file, which should also contain an overview of dependencies and which licenses they are under. For `<years>`, you should list all years in which changes were published, so if you started in a private repository in 2015, opened it up in 2016, and did the final commit in 2017, `<years>` should be 2016, 2017. For our "standard" projects, the default is to share the copyright between the eScience Center and the

PI(s) institutions, but other arrangements may have been made. So check that, and make sure everyone is represented under `<Legal entities of project partners>` .

LICENSE

The actual license of the code is stored in the *LICENSE* file. Github can add this file automatically when you create a new repository, or you can **add it via the repositories Github page**.

NOTICE

The NOTICE file is the Apache License' way of dealing with attributions. If you have any dependencies that are distributed under the Apache License, and you redistribute them (in either binary or source code form), then you must include the original NOTICE file(s) as well. If you have any attribution requirements of your own, you can add them in your own NOTICE file. If you do not distribute the dependencies, but only e.g. list them in a requirements.txt, then you do not need to include their NOTICE files in your program.

NOTICE should contain the following text, adapted with the product's name and copyright dates:

```
[PRODUCT_NAME]
Copyright [XXXX-XXXX] The Netherlands eScience Center, [PROJECT_PAR

This product includes software developed at
The Netherlands eScience Center (https://www.esciencecenter.nl/)
For the [PROJECT_NAME] project
```

If any of the software dependencies has a NOTICE file, its contents shoud be appended below. Read more in the **ASF licensing how-to**.

Modifying existing software

If you are modifying a file written by someone else, which already has its own copyright statement and Open Source license grant (possibly with a different license), then that existing statement and the grant must be kept. If you've added more than a trivial fix, add the first of the two lines above to the copyright statement, but keep the existing license grant. In these cases, we simply release our contributions under the same license the other contributors have chosen, as this avoids a lot of unnecessary complexity. If the software is proprietary, ask for advice first.

Communication

Communication to the outside world is important for visibility of Netherlands eScience Center projects and for building the user base.

Communication to other developers is a way to build community and contributors. It also increases our visibility in development world.

Home page

The software should have a homepage with all the necessary introduction information, links to documentation, source code (github) and latest release download (e.g. **github.io pages**)

The page should be created at the latest when the software is ready to be seen by the outside world. It is the place where people will learn about software, so it is important to describe its goals and functionality. It should be targeted towards non-programming users (unless software is meant for programers i.e library) but should have pointers for developers to more advanced resources (README.md)

Discussion list

Github issues, mailing list, not private email, for all project related discussions from the beginning of the project

There should be no private discussions about the project. Therefore once discussions are started (in the email), either move them to github issues or if they don't fit into issues format any more, create the mailing list.

Demo docker image in dockerhub (with Dockerfile)

When applies, ususally for services.

If software is the service Docker image should be created at the very early stage. This will allow for easier testing and platform independent use.

An online demo

Only for web applications

Online demo should be available since first stable release. When the website is the user interface for researchers, make sure there is a development version running somewhere so that they can *play around with it* and give usability feedback.

Screencast

For most software it should be possible to create a screencast. This is very useful for people to get a quick impression of what exactly you are doing without diving into the code itself. In case your software does not have a graphical user interface, even a screencast of a terminal session can be quite informative. Try to add audio, or at least subtitles, so people know what is going on in the video.

At the Netherlands eScience Center we gather screencasts in our **Youtube Channel**.

Testing

Write tests obviously takes time, so why should you do it? Test save time later on, and increase the quality of the software. More specifically:

- Makes you more confident that your software is correct.
- It saves time in finding bugs, the tests give an indication where the bug is.
- Makes it easier to make changes to the code, the tests will catch changes to way the software functions.
- Tests communicate how software is intended to function.

These points do not apply to prototype / throwaway phase.

unit tests

- **unit tests**
- **Guide: Writing Testable Code**

Continuous integration

To run testing, perform code quality analysis and build artifacts a Continuous Integration server can be used. The build will be performed every git push and pull request. Using a CI server will help with `it works for me` problems. The Netherlands eScience Center uses continuous integration services as much as possible when creating code.

continuous integration (CI), public on **Travis**

CI meaning: compile, unit test, integration test, quality analysis etc. Once there is some build process established and tests set up, CI should be configured too. It will save you a lot of time on debugging and allow for much quicker problem diagnosis.

Travis-CI

The Netherlands eScience Center public repositories should be built with **Travis-CI**. Travis-CI is free for Open Source projects. A Github repository can be added to Travis-CI by a Github user with admin right on the repository. At the moment Travis-CI performs builds in Ubuntu and OS X operating systems.

Getting started with Travis CI

PS. If you want to get mails from Travis-CI then you have to login at **https://travis-ci.org**

AppVeyor

To build repositories inside the Microsoft Windows operation system use **AppVeyor**. AppVeyor is free for Open Source projects.

Nightly builds

Most CI builds are triggered by a git push, but sometimes the repository must be build every night. Possible reasons for nightly builds:

- Make sure the repository stays working even if there are no changes pushed to the repository, but it's dependencies are changing possibly breaking the code in the repository.
- The build performs an action that needs to be performed daily like updating a cache.

For triggering nightly builds in Travis-CI **Cron jobs** can be used.

Polling tools

All major CI services support some form of cctray.xml feed. This feed can be read by polling tools to automatically keep an eye on your project builds. For instance, **BuildNotify**, **CCMenu** and **CCTray** give you a tray icon that turns red when a build fails.

Code coverage

Code coverage is a measure which describes how much of the source code is exercised by the test suite. At the Netherlands eScience Center we require minimum of 70% coverage.

Setting up code coverage for a repository depends on the programming language, see the **language specific guides** for setup instructions.

The code coverage should be performed when a test suite is run as part of Continuous Integration build job. The code coverage results can be published on code coverage and/or **code quality services**.

Code coverage services

The publishing of the code coverage can be performed during a Continuous Integration build job. The code coverage service offers a visualization of the coverage and a metric which can be displayed as a badge/shield icon on the repository website. See the **language specific guides** which code coverage services are available and preferred for that language.

Code coverage services support many languages and a usually free for Open Source projects. Below is a short list of services and their strengths.

Codecov

Shows unified coverage and separate coverage for **build matrix** e.g. different Python versions. For example project see **https://codecov.io/gh/xenon-middleware/xenon**, with a Java 7/8 and Linux/Windows/OSX OS build matrix.

Coveralls

More popular then Codecov. For example project see **https://coveralls.io/r/NLeSC/MAGMa**

End2end tests

For (web) user interfaces. **example with protractor and angular**

Once the web page has any interface, e2e tests should be implemented.

Dependencies tracking

David or other service depending on codebase language.

Checking for dependency updates should be done regularly. It can save a lot of time, avoiding code dependent on deprecated functionality.

Release

Releases are a way to mark or point to a particular milestone in software development. This is useful for users and collaborators, e.g. I found a bug running version x. For publications that refer to software, refering to a specific release enhances the reproducability.

Apache foundation describes their **release policy**.

Release cycles will depend on the project specifics, but in general we encourage quick agile development: *release early and often*

Semantic versioning

Releases are identified by a version number. **Semantic Versioning (semver)** is the most accepted and used way to add numbers to software versions. It is a way of communicating impact of changes in the software on users.

A version number consists of three numbers: major, minor, and patch, separated by a dot: *2.0.0*. After some changes to the code, you would do a new release, and increment the version number. Increment the:

- MAJOR version when you make incompatible API changes,
- MINOR version when you add functionality in a backwards-compatible manner, and
- PATCH version when you make backwards-compatible bug fixes.

Very often package managers depend on `semver` and will not work as expected otherwise.

Releasing code on github

Github makes it easy to do a release straight from your repositories website. See **github releases** for more information.

CHANGELOG.md

A change log is a way to communicate notable changes in a release to the users and contributors. It is typically a text file at the root of your repository called *CHANGELOG.md*. Every release should have relevant entry in change log.

See **Keep a CHANGELOG** for some best practices.

One command install

To not scare away users and (potential) collaborators, installing the software should be easy, a one command process. The process itself typically includes installing dependencies, compiling, testing, and finally actual installation, and can be quite complex. The use of a proper build system is strongly recommended.

Package in package manager

If your software is useful for a wider audience, create a package that can be installed with a package manager. Package managers can also be used to install dependencies quickly and easily.

- For Python use **pip**
- For Javascript use **npm**

- C, C++, Fortran, ... use packages from your distributions official repository. List your actual dependencies in the *INSTALL.md* or *README.md*

Some standard solutions for building (compiling) code are:

- The Autotools: *autoconf*, *automake*, and *libtool*. See the **Autotools Documentation**, or an **introductionary presentation by Thomas Petazzoni**
- **CMake**
- **Make**

Release quick-scan by other engineer

A check by a fellow engineer to see if the documentation is understandable? can the software be installed? etc.

Think of it as a kind of code review but with focus on mechanics, not code. The reviewer should check if: (i) there is easily visible or findable documentation, (ii) download works, (iii) there are instructions on how to (iv) install and (v) start using software, some of the things in this *scan* could be automated with continuous integration.

Citeable

Create a DOI for each release see **Making software citable**.

Dissemination

When you have a first stable release, or a subsequent major releases, let the world know! Inform your coordinator and our Communications Advisor (Lode) so we can write news item on our site, add it to the annual report, etc.

Documentation

Developed programs should be documented at multiple levels, from code comments, through API documentation, to installation and usage documentation. Comments at each level should take into account different target audience, from experienced developers, to end users with no programming skills.

Example of good documentation: **A Guide to NumPy/SciPy Documentation**

Markdown

Markdown is a lightweight markup language that allows you to create webpages, wikis and user documentation with a minimum of effort. Documentation written in markdown looks

exactly like a plain-text document and is perfectly human-readable. In addition, it can also be automatically converted to HTML, latex, pdf, etc. More information about markdown can be found here:

http://daringfireball.net/projects/markdown/

http://en.wikipedia.org/wiki/Markdown

Retext is a markdown aware text editor, that can be used to edit markdown files and convert them into HTML or PDF. It can be found at:

https://github.com/retext-project/retext

Alternatively, 'pandoc' is a command line utility that can convert markdown documents to into several other formats (including latex):

http://johnmacfarlane.net/pandoc/

An Eclipse plugin for previewing the HTML generated by markdown is available on this page:

https://marketplace.eclipse.org/content/markdown-text-editor

Readme

Clear explanation of the goal of the project with pointers to other documentation resources.

Use **GitHub flavoured markdown** for, e.g., **syntax highlighting**. (If reStructuredText or another format that GitHub renders is idiomatic in your community, use that instead.) README is targeted towards developers, it is more technical than home page. Keeping basic documentation in README.md can be even useful for lead developer, to track steps and design decisions. Therefore it is convenient to create it from the beginning of the project, when initialising git repository.

- **StackOverflow on good readme**
- **short gist with README.md template**
- **The art of README** from nodejs community

Well defined functionality

Ideally in README.md

Source code documentation

Code comments

Code comments, can be block comments or inline comments. They are used to explain what is the piece of code doing. Those should explain why something is done in the domain language and not programming language - why instead of what.

API documention

API documentation should explain function arguments and outputs, or the object methods. How they are formulated will depend on the language.

Usage documentation

- User manual (as PDF) in the "doc" directory. This is the real manual, targeted at your users. Make sure this is readable by domain experts, and not only software developers. Make sure to include:
  - Netherlands eScience Center logo.
  - Examples.
  - Author name(s).
  - Versions numbers of the software and documentation.
  - References to:
    - The eScience Center web site.
    - The project web site.
    - The Github page of the project.
    - Location of the issue tracker.
    - More information (e.g. research papers).

Documented development setup

(good example is **Getting started with khmer development**) It should be made available once there is more than one developer working on the codebase. If your development setup is very complicated, please consider providing a Dockerfile and docker image.

Contribution guidelines

Contribution guidelines make it easier for collaborators to contribute, and smooth the process of collaboration.

Guidelines should be made available once the code is available online and there is a process for contributions by other people. Good guidelines will save time of both lead developer and

contributor since things have to be explained only once. A good CONTRIBUTING.md file describes at least how to perform the following tasks:

- How to **install the dependencies**
- How to run **(unit) tests**
- What **code style** to use
- Reference to **code of conduct**
- When using a **git branching model**, the choice of branching model

An extensive example is **Angular.js's CONTRIBUTING.md**. Note that **GitHub has built in support for a CONTRIBUTING.md file**.

Code of conduct

A code of conduct is a set of rules outlining the social norms, religious rules and responsibilities of, and or proper practices for an individual. Such a document is advantagous for collaberation, for several reasons:

- It shows your intent to work together in a positive way with everyone.
- It reminds everyone to communicate in a welcoming and inclusive way.
- It provides a set of guidelines in case of conflict.

contributor covenant

CofC should be attached from the beginning of the project. There is no gain from having it with one developer, but it does not cost anything to include it in the project and will be handy when more developers join.

Documented code style

From the beginning of the project, a decision on the code style has to be made and then should be documented. Not having a documented code style will highly increase the chance of inconsistent style across the codebase, even when only one developer writes code. The Netherlands eScience Center should have a sane suggestion of coding style for each programming language we use. Coding styles are about consistency and making a choice, and not so much about the superiority of one style over the other. A sane set of guides can be found on in **google documentation**.

How to file a bug report

Describing how to properly report a bug will save a lot of developers's time. It is also useful to point users to good bug report guide like **one from Simon Tatham**

- An example of such a document for Mozilla projects
- Other example from Ubuntu Docuementation

Explained meaning of issue labels

Once users start submitting issues labels should be documented.

DOI or PID

making your code citable

Identifiers should be associated with releases and should be created together with first release.

Software citation

To get credit for your work, it should be as easy as possible to cite your software.

Your software should contain sufficient information for others to be able to cite your software, such as: authors, title, version, journal article (if there is one) and DOI (as described in the **DOI section**). It is recommended that this information is contained on a single file.

You can use the **Citation File Format** to provide this information on a human- and machine-readable format.

Read more in **the blog post by Druskat et al.**.

Print software version

Make it easy to see which version of the software is in use.

- if it's a command line tool: print version on the command line
- if it's a website: print version within the interface
- if the tool generates the output: output file should contain the version of software that generated the output

Use standards

Standard files and protocols should always be a primary choice. Using standards improves the interoperability of your software, thereby improving its usefulness.

Exchange formats

Examples include Unicode W3C, OGN, NetCDF, etc.

Protocols

Examples include HTTP, TCP, TLS, etc.

This chapter provides practical info on each of the main programming languages of the Netherlands eScience Center.

This info is (on purpose) high level, try to provide "default" options, and mostly link to more info.

Each chapter should contain:

- Intro: philosophy, typical usecases.
- Recommended sources of information
- Installing compilers and runtimes
- Editors and IDEs
- Coding style conventions
- Building and packaging code
- Testing
- Code quality analysis tools and services
- Debugging and Profiling
- Logging
- Writing documentation
- Recommended additional packages and libraries
- Available templates

Preferred Languages

At the Netherlands eScience Center we prefer Java and Python over C++ and Perl, as these languages in general produce more sustainable code. It is not always possible to choose which libraries we use, as almost all projects have existing code as a starting point.

(In alphabetical order)

- Java
- JavaScript (preferably Typescript)
- Python
- OpenCL and CUDA
- R

Selecting tools and libraries

On GitHub there is a concept of an "awesome list", that collects awesome libraries and tools on some topic. For instance, here is one for Python: **https://github.com/vinta/awesome-python**

Now, someone has been smart enough to see the pattern, and has created an awesome list of awesome lists: **https://awesome.re/**

Highly recommented to get some inspiration on available tools and libraries!

Development Services

To do development in any language you first need infrastructure (code hosting, ci, etc). Luckily a lot is available for free now.

See this list: **https://github.com/ripienaar/free-for-dev**

Java code has the big advantage of being very portable.

Recommended sources of information

- **Javadoc API Documentation**

Installing Compilers and Runtimes

Its recommended to use the latest official Oracle version (Java 8) if at all possible. OpenJDK is usually ok as well, but definitely avoid gcj.

- **Download Oracle Java**
- **Installing Oracle Java in Ubuntu (via Webupd8)**

Editors and IDEs

For Java we normally use the **Eclipse** IDE.

Coding style conventions

We follow the standard coding style defined by SUN.

Latest version seems to be the **Java Coding Style on Scribd**.

We have standard code formatting settings for eclipse.

TODO: describe tabs-vs-spaces and indentation size.

code_format_nlesc_v2.xml code_cleanup_nlesc.xml

Automated checking of the code style can be done with PMD and FindBugs.`

TODO: add (a link to) our standard ruleset.

Building and packaging code

As a build system we normally use **Gradle**. This also determines the project layout, and has standard features for packaging code.

Testing

The standard unit testing framework in Java is **JUnit**. Try to use Junit 4 if at all possible.

Use following naming scheme to distinguish unit and integration tests:

- Unit tests: **/Test*.java, **/*Test.java, and **/*TestCase.java
- Integration tests: **/IT*.java, **/*IT.java, and **/*ITCase.java

Test coverage can be measured with **Jacoco**. For running and viewing Jacoco code coverage, use **eclemma**

Code quality analysis tools and services

SonarQube

SonarQube is an open platform to manage code quality which can also show code coverage and count test results over time. SonarQube can analyze Java, C, C++, Python and Javascript. The analysis can be done in IDE or command line using **http://www.sonarlint.org/** For example project see **https://sonarcloud.io/dashboard?id=xenon-middleware_xenon-cli** Notifications of each project must be configured in your own account settings.

Codacy

Code quality and coverage grouped by file. Can setup goals to improve quality or coverage by file or category. For example project see **https://www.codacy.com/app/xenon-middleware/xenon/dashboard**

Codecov

Can show code coverages for many languages including Java, Python and Javascript. Shows unified coverage and separate coverage for matrix builds. For example project see https://codecov.io/github/xenon-middleware/xenon

Debugging and Profiling

Use jConsole or jVisualVM.

Logging

For logging, we use **the slf4j api**. The advantage of slf4j is that it is trivial to change logging implementations. The API distribution also contains a few simple implementations.

To get logging info into Eclipse, one option is to use **logback beagle**.

##Writing documentation

Java has the inbuild **JavaDoc** system for generating API documentation, usually in the form of HTML. Highly recommended.

##Recommended additional packages and libraries

JFreeChart is a Java library that allows to do nice looking charts.

Available Templates

There are currently no Java templates available. See **The Xenon repo on GitHub** as a (rather complex) example.

Distribution

We use **Bintray** to publish packages.

To make the package easy for users to install, the packages can be added to **JCenter**. JCenter is the largest repository in the world for Java and Android OSS libraries, packages and components. In a Gradle build file the JCenter repository can be used by adding:

```
repositories {
    jcenter()
}
```

Packages developed at the Netherlands eScience Center can be found in the **Bintray NLeSC repository**.

Getting started

To learn about JavaScript, view the presentations by **Douglas Crockford**:

- **Crockford on JavaScript**
- **JavaScript: The Good Parts**
- JavaScript trilogy:
  - **The JavaScript Programming Language** (1h50m)
  - **Theory of the DOM** (1h18m)
  - **Advanced JavaScript** (1h07m)

In **this video** (47m04s), Nicholas Zakas talks about sustainability aspects, such as how to write maintainable JavaScript, how to do JavaScript testing, and good programming style (much needed in JavaScript). Among others, he mentions the following style guides:

- **Google's style guide for JavaScript**;
- **Crockford's style guide** integrates with **JSLint**, which in turn is available as a plugin for Eclipse.
- Zakas has also written **an excellent book** on writing maintainable JavaScript, also within the context of working in teams. The appendix contains a style guide with explanation.

These video tutorials (totaling a couple of hours) are useful if you're just starting with learning the JavaScript language.

Another source of information for javascript, is the "web standards curriculum" made by the Web Education Community Group as part of W3C:

http://www.w3.org/community/webed/wiki/Main_Page

In particular, see the page about **Javascript best practices**

Frameworks

To develop a web application it is no longer enough to sprinkle some **JQuery** calls on a html page, a JavaScript based front end web application framework must be used. The are very many frameworks, popularity is a good way to pick one. Currently the most popular frameworks are

- **Angular**
- **React**
- **Vue.js**

All these frameworks have a command line utility to generate an application skeleton which includes the serve, build and test functionality.

Angular

Angular is a application framework by Google written in **TypeScript**.

To create a Angular application use **Angular CLI**.

React

React is a library which can used to create interactive User Interfaces by combining components. It is developed by Facebook. Where Angular and Vue.js are frameworks, including all the rendering, routing, state management functonality inside them. React only does rendering so other libraries must be used for routing and state management. **Redux** can be used to let state changes flow through React components. **React Router** can be used to navigate the application using URLs.

To create a React application use the **Create React App** How to develop the bootstrapped app further is described in the README.md.

TypeScript React Starter is a Typescript version of create react app.

Vue.js

Vue.js is an open-source JavaScript framework for building user interfaces.

To create a Vue.js application use **Vue CLI**.

TypeScript Vue Starter is a guide to write Vue applications in TypeScript.

JavaScript outside browser

Most JavaScript is run in web browsers, but JavaScript can also be run on outside browsers with **NodeJS**.

On Ubuntu (18.04) based systems, you can use the following commands to install NodeJS:

```shell
# system packages (Ubuntu/Debian)
curl -sL https://deb.nodesource.com/setup_10.x | sudo -E bash -
sudo apt-get install -y nodejs
```

NodeJS comes with a package manager called **npm**. The package manager uses **https://www.npmjs.com/** as the package repository.

Editors and IDEs

These are some good JavaScript editors:

- **Atom** by GitHub
- **Brackets** by Adobe
- **WebStorm** by JetBeans
- **Visual Studio Code** by Microsoft

The best JavaScript editors are currently WebStorm and Visual Studio Code. Atom can have some performance problems, especially with larger files.

Debugging

In web development, debugging is typically done in the browser.

- The best debugging tool suite is currently the debugger built into the Google Chrome webbrowser, and its open-source counterpart, Chromium. It can watch variables, step through the code, lets you monitor network traffic, and much more. Activate the debugger through the F12 key.
- On Firefox, use either the built-in debugging functionality (again accessible through the F12 button) or install the **Firebug** Addon for some more advanced debugging functionality.
- Microsoft has a debugging toolset called 'F12' for their Internet Explorer and Edge browsers. It offers similar capability as that of Google Chrome, Chromium, and Firefox.
- In Safari on OS X, press ⌘⌥U.

Sometimes the JavaScript code in the browser is not an exact copy of the code you see in your development environment, for example because the original source code is minified/uglified or transpiled before it's loaded in the browser. All major browsers can now deal with this through so-called *source maps*, which instruct the browser which symbol/line in a javascript file corresponds to which line in the human-readable source code. Look for the 'create sourcemaps' option when using minification/uglification/transpiling tools.

Hosting data files

To load data files with JavaScript you can't use any file system URLs due to safety restrictions. You should use a web server (which may still serve files that are local). A simple webserver can be started from the directory you want to host files with:

```bash
python3 -m http.server 8000
```

Then open the webbrowser to http://localhost:8000.

Documentation

JSDoc works similarly to JavaDoc, in that it parses your JavaScript files and automatically generates HTML documentation. The Tag Dictionary is an overview of the tag names you can use to document your code.

Testing

- Jasmine, a behavior-driven development framework for testing JavaScript code.
- Karma, Test runner, runs tests in web browser with code coverage. Use PhantomJS as headless webbrowser on CI-servers.
- Tape, a minimal testing framework that helps remove some of the black-box approach of some of the other frameworks.
- Jest, a test framework from Facebook which is integrated into the Create React App

Web based tests

To interact with web-browsers use Selenium.

Test with

- Local web browser
- Web browsers hosted by Sauce Labs, it has a matrix of web-browsers and Operating Systems. Free for open source projects.

Coding style

See general front dev guidelines and Airbnb JavaScript Style Guide.

Use a linter like eslint to detect errors and potential problems.

Showing code examples

Code examples can be stored in Gists in GitHub. bl.ocks.org allows you to view the resulting page, and serve as a small demo. There's also jsfiddle, which shows you a live preview of your web page while you fiddle with the underlying HTML, JavaScript and CSS code.

Code quality analysis tools and services

- Code climate can analyze Javascript (and Ruby, PHP). For example project see https://codeclimate.com/github/NLeSC/PattyVis
- Codacy can analyze Java, Python, Javascript and Typescript (and CSS, PHP, Scala). The analysis for Java and Python is not as good as for Javascript. The analysis is quite slow, as

it analyzes each past commit. For example project see https://www.codacy.com/app/3D-e-Chem/molviewer-tsx/dashboard

- SonarCloud is an open platform to manage code quality which can also show code coverage and count test results over time. Can analyze Java (best supported), C, C++, Python, Javascript and Typescript. For example project see https://sonarcloud.io/dashboard?id=e3dchem%3Amolviewer

TypeScript

http://www.typescriptlang.org

Typescript is a typed superset of JavaScript which compiles to plain JavaScript. Typescript adds static typing to JavaScript, which makes it easier to scale up in people and lines of code.

At the Netherlands eScience Center we prefer TypeScript over JavaScript as it will lead to more sustainable software.

Getting Started

To learn about TypeScript the following resources are available:

- youtube: tutorials playlist about TypeScript
- tutorial from Microsoft's TypeScript website
- blog post about how TypeScript can be used with the Google Chrome/Chromium debuggers (and presumably Firefox, and Internet Explorer) through the use of so-called 'source maps'. (Follow this link to set up source mapping for Firefox, also useful for debugging minified JavaScript code).
- blog post that supposedly is the definitive guide to TypeScript
- TypeScript Language Specification

Quickstart

To install TypeScript compiler run:

```shell
npm install -g typescript
```

Dealing with Types

In TypeScript, variables are typed and these types are checked. This implies that when using libraries, the types of these libraries need to be installed. More and more libraries ship with type

declarations in them so they can be used directly. These libraries will have a "typings" key in their package.json. When a library does not ship with type declarations then the libriaries `@types/<library-name>` package must be installed using npm:

```shell
npm install --save-dev @types/<library-name>
```

For example say we want to use the `react` package which we installed using `npm` :

```shell
npm install react --save
```

To be able to use its functionality in TypeScript we need to install the typings. We can search for the correct package at **http://microsoft.github.io/TypeSearch/** .

And install it with:

```shell
npm install --save-dev @types/react
```

The `--save-dev` flag saves this installation to the package.json file as a development dependency. Do not use `--save` for types because a production build will have been transpiled to Javascript and has no use for Typescript types.

Editors and IDEs

These are some good TypeScript editors:

- **Atom** by GitHub, with the `atom-typescript` Atom package.
- **Brackets** by Adobe
- **Visual Studio Code** by Microsoft
- **WebStorm** by JetBeans

The best TypeScript editors is currently Visual Studio Code as Microsoft develops both the editor and Typescript.

Debugging

In web development, debugging is typically done in the browser. Typescript can not be run directly in web browser so it must be transpiled to Javascript. To map a breakpoint in the

browser to a line in the original Typescript file **source maps** are required. Most frameworks have a project build system which generate source maps.

Documentation

It seems that **TypeDoc** is a good tool to use. Alternative could be **TSdoc**

Style Guides

TSLint is a good tool to check your codestyle.

For the **sim-city-cs project** we use **this** tslint.json file.

Python

Python is the "dynamic language of choice" of the Netherlands eScience Center. We use it for data analysis and data science projects using the SciPy stack and Jupyter notebooks, and for **many other types of projects**: workflow management, visualization, NLP, web-based tools and much more. It is a good default choice for many kinds of projects due to its generic nature, its large and broad ecosystem of third-party modules and its compact syntax which allows for rapid prototyping. It is not the language of maximum performance, although in many cases performance critical components can be easily replaced by modules written in faster, compiled languages like C(++) or Cython.

The philosophy of Python is summarized in the **Zen of Python**. In Python, this text can be retrieved with the `import this` command.

Project setup

When starting a new Python project, consider using our **Python template**. This template provides a basic project structure, so you can spend less time setting up and configuring your new Python packages, and comply with the software guide right from the start.

Use Python 3, avoid 2

Python 2 and Python 3 have co-existed for a long time, but **starting from 2020, development of Python 2 is officially abandoned**, meaning Python 2 will no longer be improved, even in case of security issues. If you are creating a new package, use Python 3. It is possible to write Python that is both Python 2 and Python 3 compatible (e.g. using **Six**), but only do this when you are 100% sure that your package won't be used otherwise. If you need Python 2 because of old, incompatible Python 2 libraries, strongly consider upgrading those libraries to Python 3 or

replacing them altogether. Building and/or using Python 2 is probably discouraged even more than, say, using Fortran 77, since at least Fortran 77 compilers are still being maintained.

- **Things you're probably not using in Python 3 – but should**
- **Six**: Python 2 and 3 Compatibility Library
- **2to3**: Automated Python 2 to 3 code translation
- **python-modernize**: wrapper around 2to3

Learning Python

- A popular way to learn Python is by doing it the hard way at **http://learnpythonthehardway.org/**
- Using `pylint` and `yapf` while learning Python is an easy way to get familiar with best practices and commonly used coding styles

Dependencies and package management

Use `pip` or `conda` (note that pip and conda can be used side by side, see also **what is the difference between pip and conda?**).

If you are planning on distributing your code at a later stage, be aware that your choice of package management may affect your packaging process. See **Building and packaging** for more info.

Pip + virtualenv

Create isolated Python environments with **virtualenv**. Very much recommended for all Python projects since it:

- installs Python modules when you are not root,
- contains all Python dependencies so the environment keeps working after an upgrade, and
- lets you select the Python version per environment, so you can test code compatibility between Python 2.x and 3.x.

To manage multiple virtualenv environments and reference them only by name, use **virtualenvwrapper**. To create a new environment, run `mkvirtualenv environment_name`, to start using it, run `workon environment_name` and to stop working with it, run `deactivate`.

If you are using Python 3 only, you can also make use of the standard library **venv** module. Creating a virtual environment with it is as easy as running `python3 -m venv /path/to/environment`. Run `. /path/to/environment/bin/activate` to start using it and `deactivate` to deactivate.

With virtualenv and venv, pip is used to install all dependencies. An increasing number of packages are using `wheel`, so pip downloads and installs them as binaries. This means they have no build dependencies and are much faster to install. If the installation of a package fails because of its native extensions or system library dependencies and you are not root, you have to revert to Conda (see below).

To keep a log of the packages used by your package, run `pip freeze > requirements.txt` in the root of your package. If some of the packages listed in `requirements.txt` are needed during testing only, use an editor to move those lines to `test_requirements.txt`. Now your package can be installed with

```shell
pip install -r requirements.txt
pip install -e .
```

The `-e` flag will install your package in editable mode, i.e. it will create a symlink to your package in the installation location instead of copying the package. This is convenient when developing, because any changes you make to the source code will immediately be available for use in the installed version.

Conda

Conda can be used instead of virtualenv and pip. It easily installs binary dependencies, like Python itself or system libraries. Installation of packages that are not using `wheel` but have a lot of native code is much faster than `pip` because Conda does not compile the package, it only downloads compiled packages. The disadvantage of Conda is that the package needs to have a Conda build recipe. Many Conda build recipes already exist, but they are less common than the `setup.py` that generally all Python packages have.

There are two main distributions of Conda: **Anaconda** and **Miniconda**. Anaconda is large and contains a lot of common packages, like numpy and matplotlib, whereas Miniconda is very lightweight and only contains Python. If you need more, the `conda` command acts as a package manager for Python packages.

Use `conda install` to install new packages and `conda update` to keep your system up to date. The `conda` command can also be used to create virtual environments.

For environments where you do not have admin rights (e.g. DAS-5) either Anaconda or Miniconda is highly recommended, since the install is very straightforward. The installation of packages through Conda seems very robust. If you want to add packages to the (Ana)conda repositories, please check **Build using conda**. A possible downside of Anaconda is the fact that this is offered by a commercial supplier, but we don't foresee any vendor lock-in issues.

Editors and IDEs

- Every major text editor supports Python, either natively or through plugins. At the Netherlands eScience Center, often used editors are **atom**, **Sublime Text** and **vim**.
- **PyDev** is an open source IDE. The source code is available in the **PyDev GitHub repository**. It has debugging, unit testing, and reporting(code analysis, code coverage) support.
- For those seeking an IDE, JetBrains **PyCharm** is the Python IDE of choice. **PyCharm Community Edition** is open source. The source code is available in the **python folder of the IntelliJ repository**. It has visual debugger, unit testing and code coverage support, profiler. JetBrains provides a **list of all tools in PyCharm**.

Coding style conventions

The style guide for Python code is **PEP8** and for docstrings it is **PEP257**. We highly recommend following these conventions, as they are widely agreed upon to improve readability. To make following them significantly easier, we recommend using a linter.

Many linters exists for Python, `prospector` is a tool for running a suite of linters, it supports, among others:

- **pycodestyle**
- **pydocstyle**
- **pyflakes**
- **pylint**
- **mccabe**
- **pyroma**

Make sure to set strictness to `veryhigh` for best results. `prospector` has its own configuration file, like the **.prospector.yml default in the Python template**, but also supports configuration files for any of the linters that it runs. Most of the above tools can be integrated in text editors and IDEs for convenience.

Autoformatting tools like `yapf` and `black` can automatically format code for optimal readability. `yapf` is configurable to suit your (team's) preferences, whereas `black` enforces the style chosen by the `black` authors. The `isort` package automatically formats and groups all imports in a standard, readable way.

Building and packaging code

To create an installable Python package, create a file `setup.py` and use the `setuptools` module. Make sure you only import standard library packages in `setup.py` , directly or through importing other modules of your package, or your package will fail to install on systems that do not have the required dependencies pre-installed. Set up continuous integration

to test your installation script. Use `pyroma` (can be run as part of `prospector`) as a linter for your installation script.

For packaging your code, you can either use `pip` or `conda`. Neither of them is **better than the other** -- they are different; use the one which is more suitable for your project. `pip` may be more suitable for distributing pure python packages, and it provides some support for binary dependencies using **`wheels`**. `conda` may be more suitable when you have external dependencies which cannot be packaged in a wheel.

- Upload your package to the **Python Package Index (PyPI)** so it can be installed with pip.
  - Either do this manually by using **twine** (**tutorial**),
  - Or configure **Travis CI** or **Circle-CI** to do it automatically for each release.
  - Additional guidelines:
    - Packages should be uploaded to PyPI using **your own account**
    - For packages developed in a team or organization, it is recommended that you create a team or organizational account on PyPI and add that as a collaborator with the owner rule. This will allow your team or organization to maintain the package even if individual contributors at some point move on to do other things. At the Netherlands eScience Center, we are a fairly small organization, so we use a single backup account (`nlesc`).
    - When distributing code through PyPI, non-python files (such as `requirements.txt`) will not be packaged automatically, you need to **add them to** a `MANIFEST.in` file.
    - To test whether your distribution will work correctly before uploading to PyPI, you can run `python setup.py sdist` in the root of your repository. Then try installing your package with `pip install dist/<your_package>tar.gz.`
- **Build using conda**
  - If desired, add packages to **conda-forge**. Use BioConda or custom channels (hosted on GitHub) as alternatives if need be.
- **Python wheels** are the new standard for **distributing** Python packages. For pure python code, without C extensions, use **`bdist_wheel`** with a Python 2 and Python 3 setup, or use **`bdist_wheel --universal`** if the code is compatible with both Python 2 and 3. If C extensions are used, each OS needs to have its own wheel. The **manylinux** docker images can be used for building wheels compatible with multiple Linux distributions. See **the manylinux demo** for an example. Wheel building can be automated using Travis (for pure python, Linux and OS X) and Appveyor (for Windows).

Testing

- **pytest** is a full featured Python testing tool. You can use it with `unittest`. **Pytest intro**
- **Using mocks in Python**
- **unittest** is a framework available in Python Standard Library. **Dr.Dobb's on Unit Testing with Python**

- doctest searches for pieces of text that look like interactive Python sessions, and then executes those sessions to verify that they work exactly as shown. Always use this if you have example code in your documentation to make sure your examples actually work.

Using `pytest` is preferred over `unittest`, `pytest` has a much more concise syntax and supports many useful features.

Please make sure the command `python setup.py test` can be used to run your tests. When using `pytest`, this can be easily configured as described in the `pytest` documentation.

Code coverage

When you have tests it is also a good to see which source code is exercised by the test suite. Code coverage can be measured with the coverage Python package. The coverage package can also generate html reports which show which line was covered. Most test runners have have the coverage package integrated.

The code coverage reports can be published online in code quality service or code coverage services. Preferred is to use one of the code quality service which also handles code coverage listed below. If this is not possible or does not fit then use one of the generic code coverage service list in the software guide.

Code quality analysis tools and services

Code quality service is explained in the The Turing Way. There are multiple code quality services available for Python. There is not a best one, below is a short list of services with their different strenghts.

Codacy

Code quality and coverage grouped by file. Can setup goals to improve quality or coverage by file or category. For example project see https://www.codacy.com/app/3D-e-Chem/kripodb/dashboard. Note that Codacy does not install your depencencies, which prevents it from correctly identifying import errors.

Scrutinizer

Code quality and coverage grouped by class and function. For example project see https://scrutinizer-ci.com/g/NLeSC/eEcology-Annotation-WS/

Landscape

Dedicated for Python code quality. Celery, Django and Flask specific behaviors. The Landscape analysis tool called `prospector` can be run locally. For example project see https://landscape.io/github/NLeSC/MAGMa

Debugging and profiling

Debugging

- Python has its own debugger called **pdb**. It is a part of the Python distribution.
- **pudb** is a console-based Python debugger which can easily be installed using pip.
- If you are looking for IDE's with debugging capabilities, please check **Editors and IDEs** section.
- If you are using Windows, **Python Tools for Visual Studio** adds Python support for Visual Studio.
- If you would like to integrate **pdb** with **vim** editor, you can use **Pyclewn**.
- List of other available software can be found on the **Python wiki page on debugging tools**.
- If you are looking for some tutorials to get started:
  - https://pymotw.com/2/pdb
  - https://github.com/spiside/pdb-tutorial
  - https://www.jetbrains.com/help/pycharm/2016.3/debugging.html
  - https://waterprogramming.wordpress.com/2015/09/10/debugging-in-python-using-pycharm/
  - http://www.pydev.org/manual_101_run.html

Profiling

There are a number of available profiling tools that are suitable for different situations.

- **cProfile** measures number of function calls and how much CPU time they take. The output can be further analyzed using the `pstats` module.
- For more fine-grained, line-by-line CPU time profiling, two modules can be used:
  - **line_profiler** provides a function decorator that measures the time spent on each line inside the function.
  - **pprofile** is less intrusive; it simply times entire Python scripts line-by-line. It can give output in callgrind format, which allows you to study the statistics and call tree in `kcachegrind` (often used for analyzing c(++) profiles from `valgrind`).

More realistic profiling information can usually be obtained by using statistical or sampling profilers. The profilers listed below all create nice flame graphs.

- vprof
- Pyflame
- nylas-perftools

Logging

- logging module is the most commonly used tool to track events in Python code.
- Tutorials:
  - Official Python Logging Tutorial
  - http://docs.python-guide.org/en/latest/writing/logging
  - Python logging best practices

Writing Documentation

Python uses **Docstrings** for function level documentation. You can read a detailed description of docstring usage in PEP 257. The default location to put HTML documentation is Read the Docs. You can connect your account at Read the Docs to your GitHub account and let the HTML be generated automatically using Sphinx.

Autogenerating the documentation

There are several tools that automatically generate documentation from docstrings. These are the most used:

- pydoc
- Sphinx (uses reStructuredText as its markup language)
  - Sphinx quickstart
  - Restructured Text (reST) and Sphinx CheatSheet
  - Instead of using reST, Sphinx can also generate documentation from the more readable NumPy style or Google style docstrings. The Napoleon extension needs to be enabled.

We recommend using Sphinx and Google documentation style. Sphinx can easily be integrated with setuptools, so documentation can be built with in the command `python setup.py build_sphinx` .

Recommended additional packages and libraries

General scientific

- NumPy
- SciPy
- Pandas data analysis toolkit
- scikit-learn: machine learning in Python
- Cython speed up Python code by using C types and calling C functions
- dask larger than memory arrays and parallel execution

IPython and Jupyter notebooks (aka IPython notebooks)

IPython is an interactive Python interpreter -- very much the same as the standard Python interactive interpreter, but with some extra features (tab completion, shell commands, in-line help, etc).

Jupyter notebooks (formerly know as IPython notebooks) are browser based interactive Python enviroments. It incorporates the same features as the IPython console, plus some extras like in-line plotting. Look at some examples to find out more. Within a notebook you can alternate code with Markdown comments (and even LaTeX), which is great for reproducible research. Notebook extensions adds extra functionalities to notebooks. JupyterLab is a web-based environment with a lot of improvements and integrated tools. JupyterLab is still under **development** and may not be suitable if you need a stable tool.

Jupyter notebooks contain data that makes it hard to nicely keep track of code changes using version control. If you are using git, you can add filters that automatically remove unneeded noise from your notebooks.

Visualization

- Matplotlib has been the standard in scientific visualization. It supports quick-and-dirty plotting through the `pyplot` submodule. Its object oriented interface can be somewhat arcane, but is highly customizable and runs natively on many platforms, making it compatible with all major OSes and environments. It supports most sources of data, including native Python objects, Numpy and Pandas.
  - Seaborn is a Python visualisation library based on Matplotlib and aimed towards statistical analysis. It supports numpy, pandas, scipy and statmodels.
- Web-based:
  - Bokeh is Interactive Web Plotting for Python.
  - Plotly is another platform for interactive plotting through a web browser, including in Jupyter notebooks.
  - altair is a *grammar of graphics* style declarative statistical visualization library. It does not render visualizations itself, but rather outputs Vega-Lite JSON data. This can lead to a simplified workflow.
  - ggplot is a plotting library imported from R.

Database Interface

- **psycopg** is an **PostgreSQL** adapter
- **cx_Oracle** enables access to **Oracle** databases
- **monetdb.sql** is **monetdb** Python client
- **pymongo** allows for work with **MongoDB** database
- **py-leveldb** are thread-safe Python bindings for **LevelDb**

Parallelisation

CPython (the official and mainstream Python implementation) is not built for parallel processing due to the **global interpreter lock**. Note that the GIL only applies to actual Python code, so compiled modules like e.g. `numpy` do not suffer from it.

Having said that, there are many ways to run Python code in parallel:

- The **multiprocessing** module is the standard way to do parallel executions in one or multiple machines, it circumvents the GIL by creating multiple Python processess.
- A much simpler alternative in Python 3 is the `concurrent.futures` module.
- **IPython / Jupyter notebooks have built-in parallel and distributed computing capabilities**
- Many modules have parallel capabilities or can be compiled to have them.
- At the eScience Center, we have developed the **Noodles package** for creating computational workflows and automatically parallelizing it by dispatching independent subtasks to parallel and/or distributed systems.

Web Frameworks

There are a lot web frameworks for Python that are very easy to run.

- **flask**
- **cherrypy**
- **Django**
- **bottle** (similar to flask, but a bit more light-weight for a JSON-REST service)

We recommend `flask`.

NLP/text mining

- **nltk** Natural Language Toolkit
- **Pattern**: web/text mining module
- **gensim**: Topic modeling

Creating programs with command line arguments

- For run-time configuration via command-line options, the built-in `argparse` module usually suffices.
- A more complete solution is `ConfigArgParse`. This (almost) drop-in replacement for `argparse` allows you to not only specify configuration options via command-line options, but also via (ini or yaml) configuration files and via environment variables.
- Other popular libraries are `click` and `fire`.

OpenCL & CUDA

Sources for learning

please add university courses and informative videos

- Parallel Reduction [**Slides**]
- GPU Memory bootcamp - Tony Scudiero [**git repo**]
  - Best Practices [**Slides**] [**Video**]
  - Beyond the Best Practices [**Slides**] [**Video**]
  - Collaborative Access Patterns [**Slides**] [**Video**]
- CUB: CUDA Collective primitives library [**Git**] [**Slides**] [**Video**]
- Best Practices Guide by PRACE [**HTML**] [**PDF**]

Documentation

- OpenCL specification [**1.2**] [**2.0**]
- CUDA Toolkit [**latest**]
  - **CUDA Programming Guide**
  - **CUDA Runtime API**

Source-to-source translation between CUDA and OpenCL

- vtsynergy (**https://github.com/vtsynergy**)
  - This was shown to work on DAS5 after copying /usr/include/limits.h to $PWD and commenting out the lines around # include_next (122-125) : "cu2cl-tool host_code.cc device_code.cu -- -DGPU_ON -I$PWD:/usr/include -I/usr/lib/gcc/x86_64-redhat-linux/4.8.2/include".
- cutocl (**https://github.com/benvanwerkhoven/cutocl**)

Overview of libraries

- OpenCL-based libraries
  - **CLBlast**
  - **clFFT**
- CUDA-based libraries
  - **cuBLAS**
  - **NVBLAS**
  - **cuFFT**
  - **nvGRAPH**
  - **cuRAND**
  - **cuSPARSE**

Foreign Function Interfaces for CUDA and OpenCL

- C++: [**Cuda**], [**OpenCL**]
- Python: [**PyCuda**], [**PyOpenCL**]
- Java: [**JCuda**], [**JOCL**]

Testing

- Unit Testing
  - Example of a unit test for CUDA kernel using the **Kernel Tuner**
- **comparing floating-point results**

Debugging and Profiling Tools

- **Nvidia Visual Profiler** [**User Guide**]
- **CUDA-GDB**
- **CUDA-MEMCHECK**

Performance Optimization

- Resources:
  - Better Performance at Lower Occupancy [**Slides**] [**Video**]
  - **Maxwell Tuning Guide**
  - **Pascal Tuning Guide**
- Generic Auto Tuners:
  - **Kernel Tuner** (Python)
  - **CLTune** (C++)

What is R?

R is a functional programming language and software environment for statistical computing and graphics: **https://www.r-project.org/**.

Philosophy and typical use cases

R is particularly popular in the social, health, and biological sciences where it is used for statistical modeling. R can also be used for signal processing (e.g. FFT), machine learning, image analyses, and natural language processing. The R syntax is similar in compactness and readability as python and matlab by which it serves as a good prototyping environment in science.

One of the strengths of R is the large number of available open source statistical packages, often developed by domain experts. For example, R-package **Seewave** is specialised in sound analyses. Packages are typically released on CRAN **The Comprehensive R Archive Network**.

A few remarks for readers familiar with Python:

- Compared with Python, R does not need a notebook to program interactively. In **RStudio**, an IDE that is installed separately, the user can run sections of the code by selecting them and pressing Ctrl+Enter. Consequently the user can quickly transition from working with scripts to working interactively using the Ctrl+Enter.
- Numbering in R starts with 1 and not with 0.

Recommended sources of information

Some R packages have their own google.group. All R functions come with documentation in a standardized format. To learn R see the following resources:

- **R for Data Science** by Hadley Wickham,
- **Advanced R** by Hadley Wickham,
- **Writing better R code** by Laurent Gatto.

Further, stackoverflow and standard search engines can lead you to answers to issues.

Getting started

Setting up R

To install R check detailed description at **CRAN website**.

IDE

R programs can be written in any text editor. R code can be run from the command line or interactively within R environment, that can be started with `R` command in the shell. To quit R environment type `q()` .

RStudio is a free powerful integrated development environment (IDE) for R. It features editor with code completion, command line environment, file manager, package manager and history lookup among others. You will have to install RStudio in addition to installing R. Please note that updating RStudio does not automatically update R and the other way around.

Within RStudio you can work on ad-hoc code or create a project. Compared with Python an R project is a bit like a virtual environment as it preserves the workspace and installed packages for that project. Creating a project is needed to build an R package. A project is created via the menu at the top of the screen.

Installing compilers and runtimes

Not needed as most functions in R are already compiled in C, nevertheless R has compiling functionality as described in the **R manual**. See **overview by Hadley Wickham**.

Coding style conventions

It is good to follow the R style conventions as **posted** by Hadley Wickham, which is seems compatible with the R style convention as posted by **Google**.

One point in both style conventions that has resulted in some discussion is the '<-' syntax for variable assignment. In the majority of R tutorials and books you will see that authors use this syntax, e.g. 'a <- 3' to assign value 3 to object 'a'. Please note that R syntax 'a = 3' will preform exactly the same operation in 99.9% of situations. The = syntax has less keystrokes and could therefore be considered more efficient and readable. Further, the = syntax avoids the risk for typos like a < -1, which will produce a boolean if 'a' exists, and a <- 1 which will produce an object 'a' with a numeric value. Further, the = syntax may be more natural for those who already use it in other computing languages.

The difference between '<-' and '=' is mainly related to scoping. See the **official R definition** for more information. The example below demonstrates the difference in behaviour:

Define a simple function named addone to add 1 to the function input:

- addone = function(x) return(x + 1)
- addone(3)
  - will produce 4
- addone(b=3)
  - will throw an error message because the function does not know argument b
- addone(b<-3)

    - will produce 4 as it will first assign 3 to b and then uses b as value for the first argument in addone, which happens to be x
- addone(x=3)
    - will produce 4 as it will assign 3 to known function argument x

The <- supporters will argue that this example demonstrates that = should be avoided. However, it also demonstrates that = syntax can work in the context of function input if = is only used for assigning values to input arguments that are expected by the function (x in the example above) and to never introduce new R objects as part of a function call (b in the example above).

From a computer science perspective it is probably best to adhere to the <- convention. From a domain science perspective it is understandable to use =. The code performs exactly the same and guarantees that new objects created as part of a function call result in an error. Please note that it is also possible to develop code with = syntax and to transfer it to <- syntax once the code is finished, the formatR package offers tools for doing this. The CRAN repository for R packages accepts both forms of syntax.

Recommended additional packages and libraries

Plotting with basic functions and ggplot2 and ggvis

For a generic impression of what R can do see: **http://www.r-graph-gallery.com/all-graphs/**

The basic R installation comes with a wide range of functions to plot data to a window on your screen or to a file. If you need to quickly inspect your data or create a custom-made static plot then the basic functions offer the building blocks to do the job. There is a **Statmethods.net tutorial with some examples of plotting options in R**.

However, externally contributed plotting packages may offer easier syntax or convenient templates for creating plots. The most popular and powerful contributed graphics package is **ggplot2**. Interactive plots can be made with **ggvis** package and embeded in web application, and this **tutorial**.

In summary, it is good to familiarize yourself with both the basic plotting functions as well as the contributed graphics packages. In theory, the basic plot functions can do everything that ggplot2 can do, it is mostly a matter of how much you like either syntax and how much freedom you need to tailor the visualisation to your use case.

Building interactive web applications with shiny

Thanks to **shiny.app** it is possible to make interactive web application in R without the need to write javascript or html.

Building reports with knitr

knitr is an R package designed to build dynamic reports in R. It's possible to generate on the fly new pdf or html documents with results of computations embedded inside.

Preparing data for analysis

There are packages that ease tidying up messy data, e.g. **tidyr** and **reshape2**. The idea of tidy and messy data is explained in a **tidy data** paper by Hadley Wickham. There is also the google group **manipulatr** to discuss topics related to data manipulation in R.

Speeding up code

As in many computing languages loops should be avoided in R. Here is a list of tricks to speed up your code:

- read.table() is sometimes faster than read.csv()
- ifelse()
- lapply()
- sapply()
- mapply()
- grep()
- %in% for testing whether and where values in one object occur in another object
- aggregate()
- which() for identifying which object indices match a certain condition
- table() for getting a frequency table of categorical data
- grep()
- gsub()
- dplyr package, see **also**

Use ?functionname to access fucntion documentation.

Package development

Building R packages

There is a great tutorial written by Hadley Wickam describing all the nitty gritty of building your own package in R. It's called **R packages**.

Package documentation

Read **Documentation** chapter of Hadleys **R packages** book for details about documenting R code.

Customary R uses `.Rd` files in `/man` directory for documentation. These files and folders are automatically created by RStudio when you create a new project from your existing R-function files.

If you use 'roxygen' function level comments starting with `#'` are recognised by `roxygen` and are used to automatically generate .Rd files. Read more about `roxygen` syntax on it's **github page**. `roxygen` will also populate `NAMESPACE` file which is necessary to manage package level imports.

R function documentation offers plenty of space to document the functionality, including code examples, literature references, and links to related functions. Nevertheless, it can sometimes be helpful for the user to also have a more generic description of the package with for example use-cases. You can do this with a `vignette`. Read more about vignettes in **Package documentation** chapter of Hadleys **R packages** book.

Available templates

- **http://rapport-package.info/**
- **http://shiny.rstudio.com/articles/templates.html**
- **http://rmarkdown.rstudio.com/developer_document_templates.html**

Testing, Checking, Debugging and Profiling

Testing and checking

Testthat is a testing package by Hadley Wickham. **Testing chapter** of a book **R packages** describes in detail testing process in R with use of `testthat`. Further, **testthat: Get Started with Testing** by Whickham may also provide a good starting point.

See also **checking** and **testing** R packages. note that within RStudio R package check and R package test can be done via simple toolbar clicks.

Continuous integration

Continuous integration can be done with for example [Travis], (**https://travis-ci.org/**), see **Chapter** on testing.

Debugging and Profiling

Debugging is possible in RStudio, see **link**. For profiling tips see **link**

Not in this tutorial yet:

- Logging

C and C++

C++ is one of the hardest languages to learn. Entering a project where C++ coding is needed should not be taken lightly. This guide focusses on tools and documentation for use of C++ in an open-source environment.

Standards

The latest ratified standard of C++ is C++17. The first standardised version of C++ is from 1998. The next version of C++ is scheduled for 2020. With these updates (especially the 2011 one) the preferred style of C++ changed drastically. As a result, a program written in 1998 looks very different from one from 2018, but it still compiles. There are many videos on Youtube describing some of these changes and how they can be used to make your code look better (i.e. more maintainable). This goes with a warning: Don't try to be too smart; other people still have to understand your code.

Practical use

Compilers

There are two main-stream open-source C++ compilers.

- **GCC**
- **LLVM - CLANG**

Overall, these compilers are more or less similar in terms of features, language support, compile times and (perhaps most importantly) performance of the generated binaries. The generated binary performance does differ for specific algorithms. See for instance **this Phoronix benchmark for a comparison of GCC 9 and Clang 7/8**.

MacOS (XCode) has a custom branch of `clang`, which misses some features like OpenMP support, and its own libcxx, which misses some standard library things like the very useful `std::filesystem` module. It is nevertheless recommended to use it as much as possible to maintain binary compatibility with the rest of macOS.

If you need every last erg of performance, some cluster environments have the Intel compiler installed.

These compilers come with a lot of options. Some basic literacy in GCC and CLANG:

- `-O` changes optimisation levels
- `-std=c++xx` sets the C++ standard used
- `-I*path*` add path to search for include files
- `-o*file*` output file
- `-c` only compile, do not link
- `-Wall` be more verbose with warnings

And linker flags:

- `-l*library*` links to a library
- `-L*path*` add path to search for libraries
- `-shared` make a shared library
- `-Wl,-z,defs` ensures all symbols are accounted for when linking to a shared object

Interpreter

There **is** a C++ interpreter called **Cling**. This also comes with a **Jupyter notebook kernel**.

Build systems

There are several build systems that handle C/C++. Currently, **the CMake system is most popular**. It is not actually a build system itself; it generates build files based on (in theory) platform-independent and compiler-independent configuration files. It can generate Makefiles, but also **Ninja** files, which gives much faster build times, NMake files for Windows and more. Some popular IDEs keep automatic count for CMake, or are even completely built around it (**CLion**). The major drawback of CMake is the confusing documentation, but this is generally made up for in terms of community support. When Googling for ways to write your CMake files, make sure you look for "modern CMake", which is a style that has been gaining traction in the last few years and makes everything better (e.g. dependency management, but also just the CMake files themselves).

Traditionally, the auto-tools suite (AutoConf and AutoMake) was *the* way to build things on Unix; you'll probably know the three command salute:

```
> ./configure --prefix=~/.local
    ...
> make -j4
    ...
> make install
```

With either one of these two (CMake or Autotools), any moderately experienced user should be able to compile your code (if it compiles).

There are many other systems. Microsoft Visual Studio has its own project model / build system and a library like Qt also forces its own build system on you. We do not recommend these if you don't also supply an option for building with CMake or Autotools. Another modern alternative that has been gaining attention mainly in the GNU/Gnome/Linux world is **Meson**, which is also based on **Ninja**.

Package management

There is no standard package manager like `pip`, `npm` or `gem` for C++. This means that you will have to choose depending on your particular circumstances what tool to use for installing libraries and, possibly, packaging the tools you yourself built. Some important factors include:

- Whether or not you have root/admin access to your system
- What kind of environment/ecosystem you are working in. For instance:
  - There are many tools targeted specifically at HPC/cluster environments.
  - Specific communities (e.g. NLP research or bioinformatics) may have gravitated towards specific tools, so you'll probably want to use those for maximum impact.
- Whether software is packaged at all; many C/C++ tools only come in source form, hopefully with **build setup configuration**.

Yes root access

If you have root/admin access to your system, the first go-to for libraries may be your OS package manager. If the target package is not in there, try to see if there is an equivalent library that is, and see what kind of software uses it.

No root access

A good, cross-platform option nowadays is to use **`miniconda`**, which works on Linux, macOS and Windows. The `conda-forge` channel especially has a lot of C++ libraries. Specify that you want to use this channel with command line option `-c conda-forge`. The `bioconda` channel in turn builds upon the `conda-forge` libraries, hosting a lot of bioinformatics tools.

Managing non-packaged software

If you do have to install a programm, which depends on a specific version of a library which depends on a specific version of another library, you enter what is called *dependency hell*. Some agility in compiling and installing libraries is essential.

You can install libraries in `/usr/local` or in `${HOME}/.local` if you aren't root, but there you have no package management.

Many HPC administrations provide **environment modules** (`module avail`), which allow you to easily populate your `$PATH` and other environment variables to find the respective package. You can also write your own module files to solve your *dependency hell*.

A lot of libraries come with a package description for `pkg-config` . These descriptions are installed in `/usr/lib/pkgconfig` . You can point `pkg-config` to your additional libraries by setting the `PKG_CONFIG_PATH` environment variable. This also helps for instance when trying to automatically locate dependencies from CMake, which has `pkg-config` support as a fallback for when libraries don't support CMake's `find_package` .

If you want to keep things organized on systems where you use multiple versions of the same software for different projects, a simple solution is to use something like `xstow` . **XStow** is a poor-mans package manager. You install each library in its own directory (`~/.local/pkg/<package>` for instance), then running `xstow` will create symlinks to the files in the `~/.local` directory (one above the XStow package directory). Using XStow in this way alows you to keep a single additional search path when compiling your next library.

Packaging software

In case you find the manual compilation too cumbersome, or want to conveniently distribute software (your own or perhaps one of your project's dependencies that the author did not package themselves), you'll have to build your own package. The above solutions are good defaults for this, but there are some additional options that are widely used.

- For distribution to root/admin users: system package managers (Linux: `apt` , `yum` , `pacman` , macOS: Homebrew, Macports)
- For distribution to any users: **Conda** and **Conan** are cross-platform (Linux, macOS, Windows)
- For distribution to HPC/cluster users: see options below

When choosing which system to build your package for, it is imporant to consider your target audience. If any of these tools are already widely used in your audience, pick that one. If not, it is really up to your personal preferences, as all tools have their pros and cons. Some general guidelines could be:

- prefer multi-platform over single platform
- prefer widely used over obscure (even if it's technically magnificent, if nobody uses it, it's useless for distributing your software)
- prefer multi-language over single language (especially for C++, because it is so often used to build libraries that power higher level languages)

But, as the state of the package management ecosystem shows, in practice, there will be many exceptions to these guidelines.

HPC/cluster environments

One way around this if the system does use `module` is to use **Easybuild**, which makes installing modules in your home directory quite easy. Many recipes (called Easyblocks) for building packages or whole toolchains are **available online**. These are written in Python.

A similar package that is used a lot in the bioinformatics community is **guix**. With guix, you can create virtual environments, much like those in Python `virtualenv` or Conda. You can also create relocatable binaries to use your binaries on systems that do not have guix installed. This makes it easy to test your packages on your laptop before deploying to a cluster system.

A package that gains more traction at the moment for HPC environments is **spack**. Spack allows you to pick from many compilers. When installing packages, it compiles every package from scratch. This allows you to be tailor compilation flags and such to take fullest advantage of your cluster's hardware, which can be essential in HPC situations

Near future: Modules

Note that C++20 will bring Modules, which can be used as an alternative to including (precompiled) header files. This will allow for easier packaging and will probably cause the package management landscape to change considerably. For this reason, it may be wise at this time to keep your options open and keep an eye on developments within the different package management solutions.

Editors

This is largely a matter of taste, but not always.

In theory, given that there are many good command line tools available for working with C(++) code, any code editor will do to write C(++). Some people also prefer to avoid relying on IDEs too much; by helping your memory they can also help you to write less maintainable code. People of this persuasion would usually recommend any of the following editors:

- Vim, recommended plugins:
  - **NERDTree** file explorer.
  - **editorconfig**
  - **stl.vim** adds STL to syntax highlighting
  - **Syntastic**
  - Integrated debugging using **Clewn**
- Emacs:
  - Has GDB mode for debugging.
- More modern editors: Atom / Sublime Text / VS Code
  - Rich plugin ecosystem
  - Easier on the eyes... I mean modern OS/GUI integration

In practice, sometimes you run into large/complex existing projects and navigating these can be really hard, especially when you just start working on the project. In these cases, an IDE can

really help. Intelligent code suggestions, easy jumping between code segments in different files, integrated debugging, testing, VCS, etc. can make the learning curve a lot less steep. Good/popular IDEs are

- CLion
- Visual Studio (Windows only, but many people swear by it)
- Eclipse

Code and program quality analysis

C++ (and C) compilers come with built in linters and tools to check that your program runs correctly, make sure you use those. In order to find issues, it is probably a good idea to use both compilers (and maybe the valgrind memcheck tool too), because they tend to detect different problems.

Automatic Formatting with clang-format

While most IDEs and some editors offer automatic formatting of files, **clang-format** is a standalone tool, which offers sensible defaults and a huge range of customisation options. Integrating it into the CI workflow guarantees that checked in code adheres to formatting guidelines.

Static code analysis with GCC

To use the GCC linter, use the following set of compiler flags when compiling C++ code:

```
-O2 -Wall -Wextra -Wcast-align -Wcast-qual -Wctor-dtor-privacy -Wdisa
-Winit-self -Wlogical-op -Wmissing-declarations -Wmissing-include-dir
-Woverloaded-virtual -Wredundant-decls -Wshadow -Wsign-conversion -Ws
-Wstrict-overflow=5 -Wswitch-default -Wundef -Wno-unused
```

and these flags when compiling C code:

```
-O2 -Wall -Wextra -Wformat-nonliteral -Wcast-align -Wpointer-arith -W
-Wmissing-prototypes -Wstrict-prototypes -Wmissing-declarations -Winl
-Wnested-externs -Wcast-qual -Wshadow -Wwrite-strings -Wno-unused-par
-Wfloat-equal
```

Use at least optimization level 2 (`-O2`) to have GCC perform code analysis up to a level where you get all warnings. Use the `-Werror` flag to turn warnings into errors, i.e. your code won't

compile if you have warnings. See this **post** for an explanation of why this is a reasonable selection of warning flags.

Static code analysis with Clang (LLVM)

Clang has the very convenient flag

```
-Weverything
```

A good strategy is probably to start out using this flag and then disable any warnings that you do not find useful.

Static code analysis with cppcheck

An additional good tool that detects many issues is cppcheck. Most editors/IDEs have plugins to use it automatically.

Dynamic program analysis using `-fsanitize`

Both GCC and Clang allow you to compile your code with the `-fsanitize=` flag , which will instrument your program to detect various errors quickly. The most useful option is probably

```
-fsanitize=address -O2 -fno-omit-frame-pointer -g
```

which is a fast memory error detector. There are also other options available like `-fsanitize=thread` and `-fsanitize=undefined` . See the GCC man page or the **Clang online manual** for more information.

Dynamic program analysis using the valgrind suite of tools

The **valgrind suite of tools** has tools similar to what is provided by the `-fsanitize` compiler flag as well as various profiling tools. Using the valgrind tool memcheck to detect memory errors is typically slower than using compiler provided option, so this might be something you will want to do less often. You will probably want to compile your code with debug symbols enabled (`-g`) in order to get useful output with memcheck. When using the profilers, keep in mind that a **statistical profiler** may give you more realistic results.

Automated code refactoring

Sometimes you have to update large parts of your code base a little bit, like when you move from one standard to another or you changed a function definition. Although this can be accomplished with a `sed` command using regular expressions, this approach is dangerous, if you use macros, your code is not formatted properly etc.... **Clang-tidy** can do these things and many more by using the abstract syntax tree of the compiler instead of the source code files to refactor your code and thus is much more robust but also powerful.

Debugging

Most of your time programming C(++) will probably be spent on debugging. At some point, surrounding every line of your code with `printf("here %d", i++);` will no longer avail you and you will need a more powerful tool. With a debugger, you can inspect the program while it is running. You can pause it, either at random points when you feel like it or, more usually, at so-called breakpoints that you specified in advance, for instance at a certain line in your code, or when a certain function is called. When paused, you can inspect the current values of variables, manually step forward in the code line by line (or by function, or to the next breakpoint) and even change values and continue running. Learning to use these powerful tools is a very good time investment. There are some really good CppCon videos about debugging on YouTube.

- GDB - the GNU Debugger, many graphical front-ends are based on GDB.
- LLDB - the LLVM debugger. This is the go-to GDB alternative for the LLVM toolchain, especially on macOS where GDB is hard to setup.
- DDD - primitive GUI frontend for GDB.
- The IDEs mentioned above either have custom built-in debuggers or provide an interface to GDB or LLDB.

Libraries

Historically, many C and C++ projects have seemed rather hestitant about using external dependencies (perhaps due to the poor dependency management situation mentioned above). However, many good (scientific) computing libraries are available today that you should consider using if applicable. Here follows a list of libraries that we recommend and/or have experience with. These can typically be installed from a wide range of **package managers**.

Usual suspects

These scientific libraries are well known, widely used and have a lot of good online documentation.

- **GNU Scientific library (GSL)**
- **FFTW**: Fastest Fourier Transform in the West

- OpenMPI. Use with caution, since it will strongly define the structure of your code, which may or may not be desirable.

Boost

This is what the Google style guide has to say about Boost:

> - **Definition:** The Boost library collection is a popular collection of peer-reviewed, free, open-source C++ libraries.
> - **Pros:** Boost code is generally very high-quality, is widely portable, and fills many important gaps in the C++ standard library, such as type traits and better binders.
> - **Cons:** Some Boost libraries encourage coding practices which can hamper readability, such as metaprogramming and other advanced template techniques, and an excessively "functional" style of programming.

As a general rule, don't use Boost when there is equivalent STL functionality.

xtensor

xtensor is a modern (C++14) N-dimensional tensor (array, matrix, etc) library for numerical work in the style of Python's NumPy. It aims for maximum performance (and in most cases it succeeds) and has an active development community. This library features, among other things:

- Lazy-evaluation: only calculate when necessary.
- Extensible template expressions: automatically optimize many subsequent operations into one "kernel".
- NumPy style syntax, including broadcasting.
- C++ STL style interfaces for easy integration with STL functionality.
- Very low-effort integration with today's main data science languages Python, R and Julia. This all makes xtensor a very interesting choice compared to similar older libraries like Eigen and Armadillo.

General purpose, I/O

- Configuration file reading and writing:
  - yaml-cpp: A YAML parser and emitter in C++
  - JSON for Modern C++
- Command line argument parsing:
  - argagg
  - Clara
- fmt: pythonic string formatting
- hdf5-cpp: The popular HDF5 binary format C++ interface.

Parallel processing

- **Intel TBB** (Threading Building Blocks): template library for task parallelism
- **ZeroMQ**: lower level flexible communication library with a unified interface for message passing between threads and processes, but also between separate machines via TCP.

Style

Style guides

Good style is not just about layout and linting on trailing whitespace. It will mean the difference between a blazing fast code and a broken one.

- **C++ Core Guidelines**
- **Guidelines Support Library**
- **Google Style Guide**
- **Google Style Guide - github** Contains the CppLint linter.

Project layout

A C++ project will usually have directories `/src` for source codes, `/doc` for Doxygen output, `/test` for testing code. Some people like to put header files in `/include` . In C++ though, many header files will contain functioning code (templates and inline functions). This makes the separation between code and interface a bit murky. In this case, it can make more sense to put headers and implementation in the same tree, but different communities will have different opinions on this. A third option that is sometimes used is to make separate "template implementation" header files.

Sustainability

Testing

Use **Google Test**. It is light-weight, good and is used a lot. **Catch2** is also pretty good, well maintained and has native support in the CLion IDE.

Documentation

Use **Doxygen**. It is the de-facto standard way of inlining documentation into comment sections of your code. The output is very ugly. Mini-tutorial: run `doxygen -g` (preferably inside a

`doc` folder) in a new project to set things up, from then on, run `doxygen` to (re-)generate the documentation.

A newer but less mature option is **cldoc**.

Resources

Online

- **CppCon videos**: Many really good talks recorded at the various CppCon meetings.
- **CppReference.com**
- **C++ Annotations**
- **CPlusPlus.com**
- **Modern C++, according to Microsoft**

Books

- Bjarne Soustrup - The C++ Language
- Scott Meyers - Effective Modern C++

Fortran

Disclaimer: In general the Netherlands eScience Center does not recommend using Fortran. However, in some cases it is the only viable option, for instance if a project builds upon existing code written in this language. This section will be restricted to Fortran90, which captures majority of Fortran source code.

The second use case may be extremely performance-critical dense numerical compute workloads, with no existing alternative. In this case it is recommended to keep the Fortran part of the application minimal, using a high-level language like Python for program control flow, IO, and user interface.

Recommended sources of information

- **Fortran90 official documentation**
- **Fortran wiki**
- **Fortran90 handbook**

Compilers

- **gfortran**: the official GNU Fortran compiler and part of the gcc compiler suite.

- **ifort**: the Intel Fortran compiler, widely used in academia and industry because of its superior performance, but unfortunately this is commercial software so not recommended. The same holds for the Portland compiler **pgfortran**

Debuggers and diagnostic tools

There exist many commercial performance profiling tools by Intel and the Portland Group which we shall not discuss here. Most important freely available alternatives are

- **gdb**: the GNU debugger, part of the gcc compiler suite. Use the **-g** option to compile with debugging symbols.
- **gprof**: the GNU profiler, part of gcc too. Use the **-p** option to compile with profiling enabled.
- **valgrind**: to detect memory leaks.

Editors and IDEs

Most lightweight editors provide Fortran syntax highlighting. Vim and emacs are most widely used, but for code completion and refactoring tools one might consider the **CBFortran** distribution of Code::Blocks.

Coding style conventions

If working on an existing code base, adopt the existing conventions. Otherwise we recommend the standard conventions, described in the **official documentation** and the **Fortran company style guide**. We would like to add the following advice:

- Use free-form text input style (the default), with a maximal line width well below the 132 characters imposed by the Fortran90 standard.
- When a method does not need to alter any data in any module and returns a single value, use a function for it, otherwise use a subroutine. Minimize the latter to reasonable extent.
- Use the intent attributes in subroutine variable declarations as it makes the code much easier to understand.
- Use a performance-driven approach to the architecture, do not use the object-oriented features of Fortran90 if they slow down execution. Encapsulation by modules is perfectly acceptable.
- Add concise comments to modules and routines, and add comments to less obvious lines of code.
- Provide a test suite with your code, containing both unit and integration tests. Both automake and cmake provide test suite functionality; if you create your makefile yourself, add a separate testing target.

Intellectual Property

As with anything else in society, some of what you can and cannot do in software development is determined by the law. Most of the constraints in this particular domain stem from intellectual property laws: laws that make abstract things like designs, stories, or computer programs resemble physical objects by allowing them to be owned.

This chapter aims to give a brief summary of relevant intellectual property laws (enough to be able to read most software licenses), explain Free and open source software licensing, and explain how combining software from different sources works from a legal perspective. It also gives some rules we have worked out to deal with common situations.

This is far from an exhaustive resource; only laws that are relevant to our software development practice (i.e. they come up regularly at the Netherlands eScience Center) are described. If you're interested in protecting a plant, boat hull, or microprocessor mask, then you should look elsewhere. Also, there are areas of law beyond intellectual property that often show up in software development practice, like contract law and consumer law; these are also not covered here.

Of course, we'll begin with a disclaimer: Good legal advice is timely, specific, and given by an expert; this chapter is none of these. It was written by an engineer, not by a lawyer, and it's a heavily simplified overview of a very complex field. The intent is to give you an overview of the basics, so that you will know when to check whether something you want to do has potential legal ramifications. Don't make any important decisions based solely on the contents of this chapter.

Executive summary

Intellectual property is a complex subject matter, and we're interested in developing code, not doing legal analysis. While we cannot always get away from doing some legal analysis in more complex cases, the majority of things we run into are relatively simple, and can be resolved by following some simple rules. This section gives such a set of rules, and does so rather conservatively, i.e. it lists only things that the eScience Center is definitely okay with. If your particular case is not listed here, then it may still be possible, but only after careful consideration. So in that case, read on and/or ask for help.

I want to publish my source code, not including any of its dependencies, is that ok?

If

- you publish your source code (and only your source code) under the Apache License version 2.0,
- *and* you do *not* include any externally-developed libraries you used,
- *and* all of the externally-developed libraries you used are under a free/open source license (see below) then you are good to go.

For the purpose of this rule, the following dependency licenses are okay

- MIT

- BSD 2-clause
- BSD 3-clause
- Apache License version 2.0
- GNU Lesser General Public License v2 or later

and any other licenses, including "for academic use only" and similar statements, are not okay.

I want to use a library with license X, is that ok?

This is certainly no problem if the library has one of the following licenses:

- MIT
- BSD 2-clause
- BSD 3-clause
- Apache License version 2.0

These are all permissive licenses that impose very little restrictions on how your program can be used. So go right ahead.

We try to avoid copyleft licenses, such as the GNU Lesser General Public License (LGPL) and GNU General Public License (GPL), but if there is no alternative available, then using a library licensed under the (L)GPL is fine too.

Rationale: The Netherlands eScience Center is a publicly funded institution, and a such we want to maximise the number of ways in which people and organisations, including commercial ones, can use the software we develop. Copyleft licenses restrict this somewhat, so we try to avoid them. However, any Free Software can still be used by anyone for any purpose, redistributed, forked, and commercialised, which is enough freedom that we will not do a lot of extra work just to avoid copyleft.

I want to publish a data set, is that ok?

If

- You or your collaborators collected the data yourselves, as part of the project,
- *and* you all agree that you want to publish it under the Creative Commons CC-BY 4.0 license

then you are good to go.

If the data set contains (possibly processed) data you obtained from elsewhere, then the licensing situation of that data needs to be evaluated first. If you or our collaborators want to use a different license, then this should be discussed first.

About the law

Laws are documents that describe what you are allowed to do in a particular jurisdiction. They are made by (hopefully democratically elected) legislators, and they're written for humans to interpret. Laws can be very specific on some points, but often also leave certain things vague.

Sometimes this is even done on purpose, when the legislators decide that they cannot foresee all the cases that will develop in the future.

In case of some conflict, either between society and some individual or company in it, or between companies or individuals, some interpretation of how the law applies to this specific case has to be made. This is done by a judge. Judges will take into account the text of the law itself, the (recorded) discussions that took place when it was made, and rulings by other judges in similar cases. By doing the latter, they try to keep things consistent and therefore fair.

The collected rulings of earlier cases are together known as case law ("jurisprudentie" in Dutch). Over time, the vague areas in a law are filled in by case law. However, this is a slow process, and it is always incomplete: if the law is vague and there is no case law yet, or no sufficiently similar case, then a gray area remains.

As a result, it often makes more sense to think about legal issues in terms of probabilities and risk, rather than in terms of truth. This means that decisions on how to act given the legal situation always have a policy component to them. How important is what you want to do, and how much risk are you willing to take?

Of course, there is always an ethical side to these kinds of decisions as well: something may be strictly speaking legal, but that doesn't automatically make it the right thing to do. While it may be impossible in some cases to say with absolute certainty whether something we want to do is legal, we should always make sure that it's the right thing to do.

Trademarks

A trademark is the exclusive right to the use of a sign or design for the purpose of identifying the manufacturer of a product or supplier of a service. Trademarks are typically words or logos, but protection may extend to colors and even smells.

Trademarks protect brands and reputations, and serve to avoid confusion in the marketplace. Because of this, similar or even identical trademarks may coexist, if the corresponding companies sell different kinds of goods or services, or operate in different areas.

As an example, Apple Records and Apple Computer can co-exist peacefully despite the similar names, as it is obvious that an Apple laptop comes from Apple Computer, and an Apple CD from Apple Records. But when Apple Computer added a sound chip to the Apple IIGS, Apple Records sued them (and later sued them again over the Mac's system sound, and then about iTunes), because they were now in the same (music) market.

Getting a trademark

Trademarks can be registered with the patent and trademark office, after which they're marked with an ® symbol. In some countries, notably the US, this in not required, and just using it in

practice to identify your products is in principle enough. Non-registered trademarks are marked with a ™ symbol.

Our Netherlands eScience Center logo is an example of a (non-registered) service mark (℠, although there is no legal protection for unregistered marks here). Service marks are essentially the same thing as trademarks, but they don't identify physical products (we don't make any) but services or intangible products, and as such are applied to equipment and uniforms and such. The idea is the same however.

Losing a trademark

Trademarks lose their protection if they no longer identify a particular manufacturer, but become general terms for a category of products. For instance, a walkman is a portable audio cassette player. Sony® owns a trademark on that word, but in 2002 an Austrian judge ruled that since the word was in the dictionary as describing any portable audio cassette player, it had become a general term that is therefore not eligible for trademark protection.

Companies do not want to lose their trademarks, so they're usually quite active about protecting them. Most companies have a trademark policy that is designed to protect their trademarks from becoming generic. Google®'s trademark policy for instance says that you should tell people to "do a Google search" for something rather than "Google it", as the latter uses the term generically to mean doing a web search. If you infringe on someone's trademark, you're likely to get a more-or-less friendly letter telling you to quit it or be sued.

Using a trademark

Using trademarked words to refer to the corresponding product or company is generally fine, just make sure that you use them together with the generic term, as in the example above. If you use a trademark, you should acknowledge that it is a trademark using one of those ubiquitous notices like "Sony® is a registered trademark of Sony Corporation". Almost all companies have rules on what to do exactly, a web search for "<company> trademark guidelines" will show you the way.

Software licenses (even Free Software licenses) typically do not give out trademark rights, so you may have to rename a fork if the origin considers your fork harmful to their brand. See e.g. Firefox® (a registered trademark of the Mozilla Foundation) and IceWeasel.

Trademark acknowledgements

Apple is a trademark of Apple, Inc., registered in the U.S. and other countries.

Firefox® and Mozilla® are registered trademarks of the Mozilla Foundation.

Google™ is a trademark of Google, Inc.

Sony® is a registered trademark of Sony Corporation.

Trade Secrets

A trade secret is a secret with an economic benefit to the company that holds it. The recipe for Coca-Cola® is an oft-cited example, the source code for a proprietary software program may be another.

Trade secrets are protected by Non-Disclosure Agreements: contracts that forbid you from sharing them with anyone. In The Netherlands, there is no specific law on trade secrets, so these contracts are all that protect them.

In particular, that means that if someone spills your trade secret, then you can sue that person, but you can't do anything against the recipient of the secret. In the US, this is different: there it is a criminal offense to make use of a leaked trade secret, and you can go to jail for doing so.

Patents

From a societal point of view, trade secrets can be considered damaging. Progress can be made much more quickly if competitors can build on each other's inventions, but that is impossible if everyone keeps their inventions a secret. Patents ("octrooien" in Dutch, "patent" means that you're looking good) are intended to remedy this situation.

A patent is the exclusive right to make, use and sell an invention, in exchange for publication of a description of it. Patents have a limited duration, which varies from place to place but is usually around 20 years. Patents cover devices that are new, inventive, and applicable to some problem. Discoveries, designs, business models, software and visualizations can not be patented (but see below).

Getting a patent

Patents are obtained by writing up a description of the patent, with a list of claims that describe the claimed invention, and submitting that description to the patent office of the country where you want protection, together with a hefty fee.

The patent office will then do a (often very cursory) check to see if the patent meets the requirements, and grant it. Once you have a patent, you are the only one allowed to use or sell the claimed invention; anyone else will need to buy a license from you, or prove that the patent is invalid when you sue them.

Software patents

While software cannot be patented because it's not a device, a computer is a device. Some time ago, clever lawyers (especially in the US) therefore started filing patents for a machine that

performs certain computational steps. While a piece of software or an algorithm therefore technically cannot be patented, anyone using that software or algorithm would still infringe the patent.

The main problem with software patents is that there are a huge number of them out there, and they're written in obfuscated legalese. Many are likely invalid due to not being new, being too obvious, or being overly broad (the patent office's checks are minimal), but defending against someone with a lot of patents is very expensive unless it's completely obvious that you're not infringing anything.

It is therefore quite easy to extort money from people by collecting a pile of patents, and threatening to sue them. Meanwhile, the benefit to society is long lost, because no one uses patents to figure out how to solve programming problems.

Unfortunately, there's not much we can do to remedy this situation. In practice, just avoid using things that you know are patented, and hope for the best.

Trademark acknowledgements

Coca-Cola® is a trademark of The Coca-Cola Company, registered in the U.S. and other countries.

Database Rights

Database rights are a very new addition to the IP stable, and they exist only in the EU and a few other countries. Database rights protect the investment made to create a particular collection of information. According to these laws, whoever invests in the creation of a database gets the exclusive right to extract or reuse (make available to others) substantial parts of the database, or repeatedly extract or reuse insubstantial parts of the database.

Getting database rights

So, if you pay someone to collect data and put it into a database, then you own the database rights on that database for the next 15 years (in the EU at least). If you then offer access to the database on a web site, people can query it and use the information they got out of it, but they're not allowed to download the entire database and share it with others. Also, making another web site that forwards queries to yours and returns the results is not allowed.

Other protections for databases

The individual data items in a database are not protected by database rights, but they may be protected by other IP laws.

For instance, if you pay someone to scan a large number of newspaper articles and put them into a database, then you get to own the database rights to that database (because you paid to make it). However, each individual article is also protected by copyright, which is owned by the newspaper. Simple facts cannot be copyrighted however, so e.g. individual measurements in a database of sensor data are not protected.

A database can also be protected by copyright, if the selection and arrangement of the contents makes it a creative work. If you manually select newspaper articles and order them in a particular way so as to tell a story, the resulting database may be eligible for copyright protection, also in places where database rights do not exist. Furthermore, the data structure of a database (e.g. the DDL description of an SQL database structure) may be protected by copyright, just like software is.

Licensing database rights

Permission to extract and reuse substantial parts of a database can be given to others by the owner of the database rights via a license. Starting with version 4.0, the well-known **Creative Commons** (CC) licenses include a grant of database rights, making them suitable for use with databases. There is also the Open Database License, which predates CC 4.0, and has a more academic origin.

The default database license at the Netherlands eScience Center is the **Creative Commons Attribution 4.0** license. Putting this license on your database will simultaneously license both the database rights and the copyright (if any) on the database itself and on its contents all under the same well-known and widely used terms.

Copyright

Copyright covers original works of authorship (works of art or science, as Dutch law puts it), like books, plays, films, music and photographs, provided there was some creativity involved in making them. Copyright also covers collections, like anthologies or coffee table books with nicely arranged photographs.

The owner of the copyright in a work has the exclusive right to copy that work, and to make derivative works.

A derivative work is itself a work, but one that depends on another work. A translation of a book is an example, because translating is itself a creative act, but the translation also derives from the original. Subtitles for TV series or a new, updated edition of a textbook are also examples of derivative works.

Getting copyrights

In any country that has signed the Berne convention on copyright, all works of authorship are automatically protected by copyright as soon as they are made. Since 1989, when the US signed the Berne convention, this goes for all major countries, but before that, there were countries where it was necessary to explicitly claim copyright on a work, by adding the © symbol or a phrase like "All rights reserved". Other than in Iraq, Somalia, North Korea and a few other such countries, this is now no longer needed, and we don't do it.

Copyrights can be transferred, e.g. by selling them or giving them away. In many countries, including in Europe, there are some rights that always remain with the author however, such as the right to be recognised as the author and to have your reputation protected with regards to the work.

A very common way in which copyright ends up in the hands of someone other than the author is by work for hire: if you make something as part of your employment, your employer gets the copyright, unless otherwise agreed.

Copyright and software

Copyright predates software, but since software is a work of authorship, it is also protected (these days most copyright laws mention it explicitly). Copyright on software covers copying of the program (in whichever form) and making derivative works.

This includes copying from disk to RAM so as to run the program. Dutch law has an explicit exception for this: if you have a legal copy on disk, then you're allowed to copy it to RAM so as to run it.

Exactly what constitutes a derivative work of a computer program or library is a gray area, with little to no case law available. In other words, no one knows for sure what a judge would decide. On the other hand, there is a kind of common understanding of how it should probably work, and people operate on those assumptions with few problems so far.

Licensing copyright

If you own the copyright for a work, including a computer program or library, then you can give others permission to make copies and derivative works by giving them a license (that's actually specifically mentioned in the law). A license is a specific or general offer of the right to make copies.

For example, Dell™ has a license from Microsoft® to make copies of Microsoft Windows® and install them on the computers they sell. This is a specific offer written down in a contract between the companies. If we put up some code on the web under an open source license, then we are making a general offer – to anyone who wants it – to use our code under those terms.

Note that the End User License Agreement that often pops up when you install software, is – despite the name – typically not a copyright license, since it doesn't give you permission to copy or create derivative works. Instead, it's legally a contract, which is why you have to click OK to accept it.

There are many software licenses out there, including some common Free and Open Source Software licenses. More on these and how to use them is in the next chapter.

Trademark acknowledgements

Dell™ is a trademark of Dell, Inc.

Microsoft® and Microsoft Windows® are either registered trademarks or trademarks of Microsoft Corporation in the United States and/or other countries.

Software licenses

Software licenses are explained in **The Turing Way** chapter.

Examples

Examples using libraries can be found in the **Turing Way software license** chapter.

Data sets: Movie review emotion

xtas contains a function that detects emotions in movie reviews. It works by fitting a model to a set of training data, and then applying the model to the xtas user's data.

The training data set it uses is available on the Internet from the website of a European university, with a note saying that it can be used for academic research purposes only. xtas automatically downloads this data set the first time the user calls the function.

Since it was created in Europe, the training data set is protected by database rights, which limit copying substantial parts of it. This means that the xtas user needs permission to have xtas download the data set, which they only have if they use the data for research purposes.

Since the download happens automatically this may not be obvious, so it is documented in the function's documentation, and the function will refuse to work unless a named argument `for_academic_research=True` is used when calling it.

xtas itself is not a database, and therefore cannot be a derivative work of the data set. The same goes for the model that is fit to the data.

An alternative way to provide this functionality would be to fit the model once, and then distribute the model (but not the data set) with xtas. Whether doing so constitutes academic

research is debatable however.

Mixed: Download a car?

For an internal research project, we needed annotated images of cars to train a neural network on. Such images can be found easily on car trading web sites, and so the question arose whether we could just grab a big collection of images from such a site.

Dutch database law contains a provision (article 5.b.) that says that retrieval of a substantial part of the contents of a database for scientific research is allowed, as long as the source is acknowledged and the use is non-commercial.

Unfortunately, this is not the only barrier. The photos on the site are also copyrighted works, owned by whoever made them, and making a copy requires their permission.

Furthermore, the web site has a set of general terms and conditions, which forbids retrieving a substantial portion of the database. These apply to anyone using the web site.

Downloading a car? Bad idea.

Trademarks: Back to the future

We have a research project on using deep learning for time series data called **mcfly**, named after the main character of the Back to the Future movies. Of course, this is a commercial franchise, so the question arose whether we can use that name for our project.

A simple name is too short to be a copyrightable work, but names can be trademarked. A trademark search revealed an English band called McFly, who have trademarked that name for the class of entertainment services. Since our research project is not in that market, this is no problem.

There is also a registered trademark for "McFly & Brown", an Amsterdam recruitment company, and that registration covers the class of "Scientific and technological services and research and design relating thereto" (even though this company does not seem to do any science or software development itself).

Of course, "McFly & Brown" is not the same as "mcfly", and the question in this case is whether the two are confusingly similar.

First, the two names are not actually the same, as we don't have the second part. Second, it seems unlikely that anyone would assume a highly technical scientific research project would be associated with a recruitment agency. Third, both names are derived from a well-known movie, which probably makes people more likely to conclude that the similarity is coincidental.

Whether any of that reasoning holds up in court we're not sure of, but it sounded reasonable enough to name the project "mcfly".

Publishing Scientific Results

Ready-to-go demos

For many projects, we will prepare attractive demos. We want to be able to show a working demo at any moment in time. Therefore, we want to have special branches in git that contain fully stand-alone demos, including a slide deck, that can just be checked out and used directly.

Handling datasets and results

Assuming you have only the software in a (private) git repo, you might want to also add and share with others the data and results related to that software:

- Add also the data and figures using git lfs (**Git Large File Storage**).
- If not, make the repo public.

Available archival / preprint servers or services

- **arXiv** (physics, mathematics, computer science, quantitative biology, quantitative finance, statistics)
- **bioRxiv** (biology)
- **PeerJ Preprints** (biological and medical sciences)
- **CogPrints** (psychology, neuroscience, linguistics, and other fields related to cognition)
- **figshare** (all disciplines)
- **GitHub** (all disciplines)
- **Social Science Research Network** (cognitive sciences, economics, humanities, law and more)

Data storage and preservation

We strongly advise to store your research data in a secure location where regular back-ups of the data are made, before you start working with the data. If it is logistically impossible to store the data in a secure location immediately after data collection then here are some tips on how to improve data preservation in the time window in between data collection and data arrival at a secure location. For example, you collect data on humans in an environment without (secure) internet connection and need to temporarily store your data offline on a laptop before being able to upload it to a data archive.

Planning data storage

We recommend that you start as early as possible to think how are you managing your data during and after your project. Some questions you should ask yourself are:

- What data am I using in my project ? Think about measurements coming from experiments (performed by you or by third parties), but also interviews, statistical information, etc.
- Where is my data coming from ? How is it being collected ?
- Where and how is this information being stored ?
- Does my data comply with the required standards applicable ? For example think of the FAIR principles, GDPR, or other ethical restrictions.

These type of considerations should usually be covered by your data management plan, if your funding agency requires so. And when it is not required by your funding agency, it is probably a good idea to have a data management plan for yourself. If you are writing a data management plan, considering using **DMPOnline**.

Tips for short term storage

Checksum and sign your data archive:

- Do a checksum on your files to check preservation of integrity. This means you will need to store the checksum somewhere, usually they are tiny, so they can be provided along with the data. In fact, some Linux distributions provide the checksum of the iso image so you can check your image when you download it. Storing checksums within the filename is not common practice anymore. A lot of data formats allow storing the checksum in the file; ie. the metadata part contains the checksum of the data part.

File permissions and location:

- If you need to work with your data, but do not plan to change it then set file access permissions to read only.
- Try to avoid processing files that are also being synced with a cloud platform (like dropbox or onedrive).
- Try to make a back-up if possible and store this back-up at a different physical location.

Specific remarks on person identifiable information:

- Do not do anything without consulting your privacy consultant.

Tips for long term storage

For long term storage we advise researchers based in The Netherlands to explore the services of **SURFsara website**, the Collaborative organization for ICT in Dutch education and research, including but not exclusively:

- **Surfdrive** for secure data sharing up to 250 GB.
- **Data archive** for long term storage of extremely large datasets.

For researchers outside the Netherlands alternative data storing platforms include:

- **https://www.re3data.org**
- **https://zenodo.org/**
- **http://rd-alliance.github.io/metadata-directory/standards/**

Making software citable

Digital Object Identifiers are globally unique identifiers which can point to any digital object, such as a version of a paper, a version of software etc. This has the advantage that it is unambigous and standardized. For papers, using DOIs is commonplace, and a DOI is usually provided by the publisher. For software, you can make your own DOI with **Zenodo**:

1. You can tell people how to cite your software by including a `CITATION.cff` file in the root of your repository (You can read up on the rationale of `CITATION.cff` files in **this blog**). However, writing `CITATION.cff` files by hand is a bit tedious and error-prone, so instead go to **https://citation-file-format.github.io/cff-initializer-javascript/** and fill in the provided web form.

2. Make a **Zenodo** account and link it with your GitHub account as explained on **guides.github.com/activities/citable-code**.

3. You can tell Zenodo what metadata you want to associate with the software by including a `.zenodo.json` file in the root of your repository, but writing that file by hand is also error-prone. Therefore it is advisable to just generate it from the `CITATION.cff` file. To do so, you'll need a command line tool `cffconvert` which you can install **from PyPI** by:

   bash
   ```
   pip install --user cffconvert
   ```

4. Make sure that your `CITATION.cff` is valid YAML by copy-pasting the contents to **http://www.yamllint.com/**.

5. Make sure that your `CITATION.cff` is valid CFF, by:

   bash
   ```
   # (in the repository's root directory)
   cffconvert --validate
   ```

If the command does not return anything, that means the CFF is valid.

6. Generate the `.zenodo.json` file using `cffconvert` as follows:

```bash
cffconvert --ignore-suspect-keys --outputformat zenodo --outfile
```

7. On Zenodo, make sure to 'Flip the switch' to the `on` position on the GitHub repository that you want to make a release of.

8. Go to your Github repository, use the *Create a new release* button to create a release on GitHub.

9. Zenodo should automatically be notified and should make a snapshot copy of the current state of your repository (just one branch, without any history), and should also assign a persistent identifier (DOI) to that snapshot.

   when things don't work

   In case the GitHub-Zenodo integration does not work as expected, there are two places to go and look for information:

   1. On GitHub:
      - go to `https://github.com/<org>/<repo>/`
      - select `Settings`
      - select `Webhooks`
      - select select the Zenodo webhook (may require GitHub login)
      - scroll down to `Recent deliveries`
      - click on one of the listed deliveries for details on the request, the response, and to request redelivery.
   2. On Zenodo:
      - go to `https://zenodo.org/account/settings/github/`
      - select the repository that you want to see the diagnostic information of
      - click on one of the releases to see the *Payload* Zenodo received from GitHub, as well as the *Metadata* that Zenodo has associated with your release, or *Errors* if there were any.

10. Use the DOI whenever you refer to your software, be it in papers, posters, or even tweets and blogs.

11. Add the software's Zenodo badge to your repository's README.

#e-Science Conferences, Journals, and Workshops

This is a list of Conferences, Journals, and Workshops related to eScience.

Conferences

- The IEEE International Conference on eScience Yearly (computer science) conference on eScience.
- UK Conference of Research Software Engineers.
- German Conference of Research Software Engineers.
- The European Geosciences Union General Assembly (EGU) has a track by the Division on Earth and Space Science Informatics (ESSI).
- Free Open Source Software for GeoInformatics
- International Conference on Computational Science
- PASC Conference

There is also a community page with a list of upcoming events on the eScience Center website.

Journals

- SoftwareX.
- Journal of Open Research Software.
- Journal of Open Source Software

See A list at the Software Suistainability Institute.

Workshops

Contributing

This Knowledge Base is primarily written by the eScience Research Engineers at the Netherlands eScience Center. The intended audience is anyone interested in eScience and research software development in general or how this is done at the eScience Center specifically.

Scope

To make sure the information in this knowledge base stays relevant and up to date it is intentionally low on technical details. The Knowledge base contains information on the process we use to do projects and develop software.

Workflow for making contributions

Contributions by anyone are most welcome.

Please use branches and pull requests to contribute content. If you are not part of the Netherlands eScience Center organization but would still like to contribute please do by submitting a pull request from a fork.

```shell
git clone https://github.com/NLeSC/guide.git
git branch newbranch
git checkout newbranch
```

Add your new awesome feature, fix bugs, make other changes.

To view changes locally, host the repo with a static file web server.

```shell
python3 -m http.server 4000
```

To view the documentation in a web browser (default address: **http://localhost:4000**):

To check if there are any broken links using **liche** in a Docker container:

```shell
docker run -v $PWD:/docs peterevans/liche:1.1.1 -t 60 -c 16 -d /docs
```

If everything works as it should, `git add`, `commit` and `push` like normal.

If you have made a significant contribution to the guide, please make sure to add yourself to the `CITATION.cff` file so your name can be included in the list of authors of the guide.

Chapter Owners

To see who is responsible for which part of the guide see **chapter_owners.md**.

Chapter Owners

This is a list of who is responsible for which part of the guide.

Overall Maintainer: Bouwe Andela

- Introduction: Jason Maassen

- Software Development:
  - Overall: Jason Maassen
  - Code Review: Lourens Veen
- Language Guides:
  - Introduction: Jason Maassen
  - Java: Christiaan Meijer
  - JavaScript and TypeScript: Jurriaan Spaaks
  - Python: Patrick Bos
  - OpenCL and CUDA: Ben van Werkhoven
  - R: Vincent van Hees
  - C and C++: Johan Hidding and Patrick Bos
  - Fortran: Gijs van den Oord
- Intellectual Property: Lourens Veen
- Publishing Scientific Results: Willem van Hage
- Access to e-Infrastructure: Jason Maassen
- Projects: Jisk Attema
- Contributing to this Guide: Jason Maassen

Access to (Dutch) e-Infrastructure

To successfully run a project and to make sure the project is sustainable after it has ended, it is important to choose the e-Infrastructure carefully. Examples of e-Infrastructure used by eScience Center projects are High Performance Computing machines (Supercomputers, Grids, Clusters), Clouds, data storage infrastructure, and web application servers.

In general PI's will already have access to (usually local) e-Infrastructure, and are encouraged to think about what e-Infrastructure they need in the project proposal. Still, many also request our help in finding suitable e-Infrastructure during the project.

Which infrastructure is best very much depends on the project, so we will not attempt to describe the optimal infrastructure here. Instead, we describe what is most commonly used, and how to gain access to this e-Infrastructure.

Lack of e-Infrastructure should never be a reason for not being able to to a project (well). If you ever find yourself without proper e-Infrastructure, come talk to the Efficient Computing team. We should be able to get you going quickly.

SURF

SURF is the most obvious supplier of e-Infrastructure for Netherlands eScience Center projects. For all e-Infrastructure needs we usually first look to SURF. This does not mean SURF is our exclusive e-Infrastructure provider. We use whatever infrastructure is best for the project, provided by SURF or otherwise.

Getting access to SURF infrastructure

In general access to SURFsara resources is free of charge for scientists in The Netherlands. For most infrastructure gaining access is a matter of filling in a simple web-form, which you can do yourself on behalf of the scientists in the project. Exceptions are the Cartesius and Lisa, for which a more involved process is required. For these machines, only the PI of a project can submit (or anyone else with an NWO Iris account).

The Netherlands eScience Center also has access to the infrastructure provided by SURFnet. Access is normally done on a per-organization basis, so may vary from one project partner to the next.

Available systems at SURF

Here we list some of the most likely to be used resources at SURF. See the **overview of all SURF services and products**, and **detailed information on the SURFsara infrastructure**.

SURFsara:

- **Cartesius**: The national supercomputer of The Netherlands. It contains a lot of very high performance machines, connected through a fast interconnect (about 41000 cores in total, plus 132 GPUs). It also has a large storage system (7+ Pb). Cartesius is typically designed for large parallel applications that require thousands of cores at once.
- **Lisa**: National Cluster. Similar machines as the Cartesius, without the interconnect (about 8000 cores in total). Storage also more limited. Lisa is typically designed to run lots of small (1 to 16 core) applications at the same time.
- **Grid**: Same machines again, now with a Grid Middleware. Not recommended for use in eScience Center projects.
- **HPC Cloud**: On demand computing infrastructure. Nice if you need longer running services, or have a lot of special software requirements.
- **Hadoop**: Big Data analytics framework.
- **BeeHub**: Lots of storage with a webDAV interface.
- **Elvis**: Remote rendering cluster. Creates a remote desktop session to a Linux machine with powerful Nvidia Graphics installed.
- **Data Archive**: Secure, long-term storage of research data on tape. Access to archive included with Cartesius and Lisa project accounts.

SURFnet:

- **SURFconext**: Federated identity management. Allows scientists to login to services using their home organization account. Best known example is SURFspot. Can be added to custom services as well.
- **SURFdrive**: Dropbox-like service hosted by SURF.

Ask questions to: **helpdesk@surfsara.nl**.

DAS-5

The Netherlands eScience Center participates in the **DAS-5 (Distributed ASCI Supercomputer)**, a system for experimental computer science. Though not intended for production work, it is great for developing software on, especially HPC, parallel and/or distributed software.

DAS-5 consists of 6 clusters at 5 different locations in the Netherlands, with a total of about 200 machines, over 3000 cores, and about 800Tb total storage. These clusters are connected with dedicated lightpaths. Internally, each cluster has a fast interconnect. DAS-5 also contains an ever increasing amount of accelerators (mostly GPU's).

DAS-5 is explicitly meant as an experimentation platform: any job should be able to run instantly, long queue times should be avoided. Running long jobs is therefore not allowed during working hours. During nights and weekends these rules do not apply. See **the usage policy**.

Any eScience Center employee can get a DAS-5 account, usually available within a few hours.

Security and convenience when committing code to GitHub from a cluster

When accessing a cluster, it is generally **safer to use a pair of keys than to login using a username and password**. There is a **guide on how to setup those keys**. Make sure you encrypt your private key and that it is not automatically decrypted when you login to your local machine. Make a separate pair of keys to access your GitHub account following **GitHub's instructions**. It involves **uploading your public key to your GitHub account** and **testing your connection**.

When committing code from a cluster to GitHub, one needs to store an encrypted private key in the $HOME/.ssh directory on the cluster. This is inconvenient, because it requires submitting a password to unlock the private key. This password has to be resubmitted when SSHing to a local node from the head node. To bypass this inconvenience **SSH agent forwarding** is recommended. It is very simple. On your local machine, make a $HOME/.ssh/config file to contain the following:

```
Host example.com
    ForwardAgent yes
```

Replace example.com by the head node of your cluster, i.e. the node you use to login to. Next,

```
chmod 600 $HOME/.ssh/config.
```

Done!

The only remaining problem is that SSH keys cannot be used when git cloning was done using https instead of SSH, but that can be **corrected**:

```
git remote set-url origin git@github.com:username/repo.git
```

Commercial Clouds

If needed a project can use commercial cloud resources, normally only if all SURF resources do not meet the requirements. As long as the costs are within limits these can come out of the eScience Center general project budget, for larger amounts the PI will need to provide funding.

We do not have an official standard commercial cloud provider, but have the most experience with Amazon AWS.

Procolix

If a more long term infrastructure is needed which cannot be provided by SURF, the default company we use for managed hosting is **Procolix**. Procolix hosts our eduroam/surfconext authentication machines.

In principle the eScience Center will not pay for infrastructure needed by projects. In these cases the PIs will have to pay the bill.

GitHub Pages

If a project is in need of a website or webapp using only static content (javascript, html, etc), it is also possible to host this at github. See **https://pages.github.com/**

Local Resources

A scientist may have access to locally available infrastructure.

Other

This list does not include any resources from Nikhef, CWI, RUG, Target, etc, as these are (as far as we know) not open to all scientists.

Avoid if possible

Try to avoid using self-managed resources (the proverbial machine under the Postdoc's desk). This may seem an easy solution at first, but will most probably require significant effort over the course of the project. It also increases the changes of the infrastructure disappearing at some random moment after the project has finished.

DAS-5

This text gives a couple of practical hints to get you started using the DAS-5 quickly. It is intended for people with little to no experience using compute clusters.

First of all, and this is the most important point in this text: read the usage policy and make sure you understand every word of it: **http://www.cs.vu.nl/das5/usage.shtml**

The DAS-5 consists of multiple cluster sites, the largest one is located at the VU, which you can reach using by the hostname `fs0.das5.cs.vu.nl` . The firewall requires that your IP is whitelisted, which means you will be able to access the DAS from the eScience Center office, but not directly when you are somewhere else. To use the DAS from anywhere you can use eduVPN.

When you login in it means you are logged into the headnode, this node should not be used for any computational work. The cluster uses a reservation system, if you want to use any node that is not the head node, you must use the reservation system to gain access to a compute node. The reserveration system on DAS-5 is called Slurm, you can see all running jobs on the cluster using `squeue` and cancel any of your running jobs with `scancel <jobid>` .

The files in your home directory `/home/username/` will be backed up automatically, if you accidently delete an important file you can email the maintainer and kindly request him to put back an old version of the file. If you have to store large data sets put them under `/var/scratch/username/` , the scratch space is not backed up.

You can use the command `module` to gain access to a large set of preinstalled software. Use `module list` to see what modules are currently loaded and `module avail` to see all

available modules. You can load or unload modules with the 'module load' and `module unload`. You may want to add some of the modules you frequently use to your bashrc. Note that all that these modules do is add or remove stuff from your `PATH` and `LD_LIBRARY_PATH` environment variables. If you need software that is not preinstalled, you can install it into your home directory. For installing Python packages, you have to use Anaconda or `pip install --user`.

If you want an interactive login on any of the compute nodes through the reservation system, you could use: `srun -N 1 --pty bash`. The srun command is used to run a program on a compute node, -N specifies the number of nodes, --pty specifies this is an interactive job, bash is the name of the program being launched. This reservation is only cancelled when you logout of the interactive session, please observe the rules regarding reservation lengths.

To access the nodes you've reserved quickly it's a good idea to generate an ssh key and add your own public key to your 'authorized_keys' file. This will allow you to ssh to nodes you have reserved without password prompts.

To reserve a node with a particular GPU you have to specify to srun what kind of node you want. I have the following alias in my bashrc, because I use it all the time:
`alias gpurun="srun -N 1 -C TitanX --gres=gpu:1"`
If you prefix any command with `gpurun` the command will be executed on one of the compute nodes with an Nvidia GTX Titan X GPU in them. You can also type `gpurun --pty bash` to get an interactive login on such a node.

Running Jupyter Notebooks on DAS-5 nodes

If you have a Jupyter notebook that needs a powerfull GPU it can be useful to run the notebook not on your laptop, but on a GPU-equipped DAS-5 node instead.

How to set it up

It can be a bit tricky to get this to work. In short, what you need is to install jupyter, for example using the following command:

```
pip install jupyter
```

And it's recommended that you add this alias to your .bashrc file:

```
`alias notebook-server="srun -N 1 -C TitanX --gres=gpu:1 bash -c 'hos
```

Now you can start the server with the command `notebook-server`.

You just need to connect to your jupyter notebook server after this. The easiest way to do this is to start firefox on the headnode (fs0) and connect to the node that was printed by the `notebook-server` command. Depending on what node you got from the scheduler you can go to the address `http://node0XX:8888/`. For more details and different ways of connecting to the server see the longer explanation below.

More detailed explanation

First of all, you need to install jupyter into your DAS-5 account. I recommend using miniconda, but any Python environment works. If you are using the native Python 2 installation on the DAS don't forget to add the `--user` option to the following pip command. You can install Jupyter using: `pip install jupyter`.

Now comes the tricky bit, we are going to connect to the headnode of the DAS5 and reserve a node through the reservation system and start a notebook server on that node. You can use the following alias for that, I suggest storing it in your .bashrc file:

```
alias notebook-server="srun -N 1 -C TitanX --gres=gpu:1 bash -c 'hostname; XDG_RUNTIME_DIR= jupyter notebook --ip=* --no-browser'"
```

Let's first explain what this alias actually does for you. The first part of the command is similar to the `gpurun` alias explained above. If you do not require a GPU in your node, please remove the `-C TitanX --gres=gpu:1` part. Now let's take a look at what the rest of this command is doing.

On the node that we reserve through `srun` we execute the following bash command:

```
hostname; XDG_RUNTIME_DIR= jupyter notebook --ip=* --no-browser'
```

This is actually two commands, the first only prints the name of the host, which is important because you'll need to connect to that node later. The second command starts with unsetting the environment variable XDG_RUNTIME_DIR.

On the DAS, we normally do not have access to the default directory pointed to by the environment variable XDG_RUNTIME_DIR. The Jupyter notebook server wants to use this directory for storing temporary files, if XDG_RUNTIME_DIR is not set it will just use /tmp or something for which it does have permission to access.

The notebook server that we start would normally only listen to connections from localhost, which is the node on which the notebook server is running. That is why we pass the `--ip=*` option, to configure the notebook server to listen to incoming connections from the headnode. Be warned that this is actually highly insecure and should only be used within trusted environments with strict access control, like the DAS-5 system.

We also need the `--no-browser` no browser option, because we do not want to run the browser on the DAS node.

You can type `notebook-server` now to actually reserve a node and start the jupyter notebook server.

Now that we have a running Jupyter notebook server, there are 2 different approaches to connect to our notebook server:

1. run your browser locally and setup a socks proxy to forward your http traffic to the headnode of the DAS
2. starting a browser on the headnode of the DAS and use X-forwarding to access that browser

Approach 1 is very much recommended, but if you can't get it to work, you can defer to option 2.

Using a SOCKS proxy

In this step, we will create an ssh tunnel that we will use to forward our http traffic, effectively turning the headnode of the DAS into your private proxy server. Make sure you that you can connect to the headnode of the DAS, for example using a VPN. The following command is rather handy, you might want to save it in your bashrc:

```
alias dasproxy="ssh -fNq -D 8080 <username>@fs0.das5.cs.vu.nl"
```

Do not forget to replace `<username>` with your own username on the DAS.

Option `-f` stands for background mode, which means the process started with this command will keep running in the background, `-N` means there is no command to be executed on the remote host, and `-q` stands for quiet mode, meaning that most output will be surpressed.

After executing the above ssh command, start your local browser and configure your browser to use the proxyserver. Manually configure the proxy as a "Socks v5" proxy with the address 'localhost' and port 8080.

After changing this setting navigate to the page `http://node0XX:8888/` , where `node0XX` should be replaced with the hostname of the node you are running the notebook server on. Now in the browser open your notebook and get started using notebooks on a remote server!

Using X-Forwarding

Using another terminal, create an `ssh -X` connection to the headnode of the DAS-5. Note that, it is very important that you use `ssh -X` for the whole chain of connections to node, including the one used to connect to the headnode of the DAS and any number of intermediate servers you are using. This also requires that you have an X server on your local machine, if you are running Windows I recommend installing VirtualBox with a Linux GuestOS.

On the headnode type `firefox http://node0XX:8888/` , where `node0XX` should be replaced with the hostname of the node you are running the notebook server on. Now in the browser open your notebook and get started using notebooks on a remote server!

Projects

The Netherlands eScience Center is a projects based organization. Projects are done in partnership with scientists, usually from a Dutch University.

new Project()

There are several ways a new project gets initiated at the Netherlands eScience Center. In general, projects are started via one of our project calls. See **https://www.esciencecenter.nl/project-calls** for more information.

Kickoff Meeting

Each project starts with a kickoff meeting at the Netherlands eScience Center. At this meeting the PI, eScience engineer, Coordinator, and an MT-member are present. Other project partners are welcome.

For this meeting the standard agenda is:

- Round of introductions.
- Assignment of the eScience engineer(s) and coordinator.
- Netherlands eScience Center introduction presentation (by coordinator).
- Project introduction (by PI).
- Discussion on initial project planning and deliverables.
- Any other business.

In the Netherlands eScience Center introduction presentation several important topics are explained:

- How do we work.
- What is the role of the eScience engineer and coordinator.
- Project life cycle (annual reviews and rapports, payment, project end, etc.).
- How to communicate with the Netherlands eScience Center.
- Publications.
- Intellectual property (IP).
- Communication by the Netherlands eScience Center (project page at eScience Center website, pitches, etc.).
- Software and software quality.
- Role of eStep, knowledge base, etc.

Project Planning

Project Reviews

For all project longer than a year (typically full projects and alliances), the MT organizes annual reviews. The details are described in Section 9.4.2 of the protocol document. The annual reviews are organized and chaired by an MT member, and the PI, eScience engineer(s), eScience coordinator, and other partners (posdocs, co-PIs, etc) are present.

The goals are as follows:

- Progress of project relative to planning.
- Innovation, research, deliverables, eStep.
- Identify key success stories/messages to share with key opinion formers.
- Ensure efficient use of engineer resources, identify bottlenecks and areas to improve.
- Financial status.
- Look for ways to extend collaborations.
- Consider project legacy and post-funding support.
- Potential interaction with other eScience Center projects.

The standard agenda for this 1.5 hour meeting is:

- Presentation by the PI (20 minutes)
- Presentation eScience Engineer (20 minutes) including description of role and deliverables.
- Discussion (40 minutes)
- Summary, action points and conclusions.

Communication

Pitch presentation (1 to 3 slides)

Pitch presentation should be prepared, and updated on a regular basis.

End of a Project

End-of-project document

Project proposals are focused on their scientific domain, and are not always clear on the necessary escience. Also, during a project the escience requirements can change, and its actual escience component can be different from the originally proposed methods and tools. A final project report will focus on the scientific domain (published papers) and financial accounting. All in all, this leaves the escience part of projects a bit undocumented. Therefore, we could use a small informal document, for internal use, describing the project from the perspective of an escience engineer. In principle the escience is shared with the engineer and coordinator, and is discussed during the project reviews. Any reusable software is added to eStep, or to the knowledgebase. This document can therefore be high-level and short. It is meant to facilitate re-using tools and techniques for other escience projects, provide (links to) information and

background material for escience presentations / PR, and provide a possible starting point for continuation of the project.

As this kind of documentation is only valuable if engineers can freely share their opinions and experiences (also negative ones!), this document itself is not meant for external distribution.

Contents

- high-level description of actual escience requirements in the project
- what went great, what could have gone better
- pointers (URL) to project documentation
- motivation for chosen approach,
- high-level description of used or developed tools and references to them (github, website,)
- eScience presentation for (re-)use in the form of a powerpoint document (so the images, text, and or slides can be extracted). Check the pitch and project presentations to see if they are sufficient, ask coordinator.

written by

- escience engineer(s) working on the project

target audience

- escience engineers
- escience coordinators

schedule

- should be written during the last weeks of the project
- Stored on the internal sharepoint site

Support

The Netherlands eScience Center provides very limited support for software. During a project we make every effort to create low-maintenance code by building on as many standard components as possible, using software from eStep, and putting a lot of effort into documenting and testing software. Also, by using standard file formats and API's we try to limit the effort required to maintain software, and make it easier to continue development.

After a project has finished the eScience Center will in principle not further support the software. Reported bugs in our own software will of course have a high chance of being looked at, but

this also has its limits. We cannot in any way contribute to the administration of infrastructure needed after a project has ended.

Because of the lack of support after projects is is a good idea to start to think about and make agreements on where software will land and who will maintain infrastructure at the very beginning of a project. The project proposal should already contain a plan.

For in-house developed eStep software we *do* provide some support, though even here only limited time is available for this. See the technology page on the website (**https://www.esciencecenter.nl/technology**) for the list of supported software.

Software checklist

This section contains a list of items which are required to help software reach a sufficient quality standard. The following list of items links to explanation in other sections of this chapter.

The **checklist matrix** provides an indication of which items are important at different development stages.

Version control

- **version control from the beginning of the project**
- **use git as version control system (vcs)**
- **choose one branching model**
- **public vcs repository**
- **meaningful commit messages**

Releases

- **semantic versioning**
- **tagged releases**
- **CHANGELOG.md**
- **one command install**
- **package in package manager**
- **discuss release cycle with coordinator**
- **release quick-scan by other engineer**
- **Dissemination**

Licensing

- **Apache 2 license**
- **compatible license of all libraries**

- NOTICE(.txt|.md)

Communication

- home page
- discussion list
- demo docker image in dockerhub (with Dockerfile)
- an online demo
- screencast

Code Quality

- use editorconfig
- code style applied in automated way

Testing

- unit tests
- continuous integration
- continuous code coverage
- end2end test
- dependencies tracking

Documentation

- README.md
- well defined functionality
- source code documentation
- usage documentation
- documented development setup
- contribution guidelines
- code of conduct
- documented code style
- how to file a bug report
- explained meaning of issue labels
- DOI or PID
- CITATION.cff file
- print software version

Standards

- Exchange formats
- Protocols

Checkmatrix for 'eStep friendly' projects.

This matrix shows what parts of the software sustainability checklist should be taken care of at (perhaps slightly before) what state of a project.

Though very generic in scope and context, this is an eScience Center specific list. This allows us to keep the number of "states" low.

Explanation of project states

- ***Prototype phase***. The first step in most software development is trying out different things with no intention in keeping the intermediate results. Signs you could be in this phase:
  - You switch programming languages.
  - You throw away all of your code once in a while
  - You work on the code by yourself
  - You are waiting with showing other people your code until you "clean it up a bit first".
- ***Pre-release phase***. Eventually you get software you intend to keep. Signs you could be in this phase:
  - You have multiple developers.
  - You have external contributors.
  - You are working up to a release.
  - Users ask you if the software is done yet.
- ***Maturity phase***: Software that has reached maturity, has a clear function and scope, and is used. Signs you could be in this phase:
  - The software has a release.
  - The software has users: people actually using your software/code
  - You have external contributor
  - The software is actively used and contributed to by so many people that it becomes a community project rather than an eScience Center project.

These states happen in order and are exclusive.

Version Control

Item / Phase	Prototype	Pre-release	Mature
use git as version control system (vcs)	X		
use GitHub flow branching model (use feature branches and pull requests)		X	
public vcs repository (github)	X		

Item / Phase	Prototype	Pre-release	Mature
meaningful commit messages	X		

Releases

Item / Phase	Prototype	Pre-release	Mature
semantic versioning			X
tagged releases (github releases)			X
CHANGELOG.md (Keep a CHANGELOG)			X
one command install (pip, npm etc)			X
package in package manager (pypi, npm etc)			X
discuss release cycle with coordinator		X	
release quick-scan by other engineer (is documentation understandable, can it be installed, etc)			X
notify Lode for dissemination (news item on site / annual report, etc)			X

Licensing

Item / Phase	Prototype	Pre-release	Mature
Apache 2 license	X		
compatible license of all libraries	X		
`NOTICE(.txt or .md)` listing licenses, request citation of paper if applicable	X		

Communication

Item / Phase	Prototype	Pre-release	Mature
home page with all the necessary introduction information, links to documenation, source code (github) and latest release download (eg. github.io pages)			X
project discussion list (github issues, mailing list, not private email) for all project related discussions from the beginning of the project	X		

Item / Phase	Prototype	Pre-release	Mature
for services: a demo docker image in dockerhub (with Dockerfile)			X
for websites: an online demo			X
Pitch presentation (1 to 3 slides)		X	
Few sentences about the project for **the technology pages on our website**			X

Testing

Item / Phase	Prototype	Pre-release	Mature
unit tests		X	
build tests		X	
continuous integration, public on **Travis**		X	
continuous code coverage and code quality metrics public, minimum 70% coverage required			X
end2end test for (web) user interfaces			X
track dependencies (with **David** or other service depending on codebase language)			X

Documentation

Item / Phase	Prototype	Pre-release	Mature
`README.md` - clear explanation of the goal of the project with pointers to other documentation resources. Use **GitHub flavored markdown** for, e.g., **syntax highlighting**.	X		
well defined functionality		X	
source code documentation		X	
usage documentation		X	
documented development setup (good example is **Getting started with khmer development**)		X	
contribution guidelines **egzample**		X	
code of conduct (**contributor covenant**)		X	
documented code style		X	
meaning of issue labels used		X	

Item / Phase	Prototype	Pre-release	Mature
DOI or PID (making your code citable)			X

Development setup

Item / Phase	Prototype	Pre-release	Mature
using the eScience Center coding style is required		X	
editorconfig		X	
applied code style in automated way if possible (i.e using linters and code formaters)		X	
dev environment docker images in Dockerhub (with Dockerfile)		X	

Use standards

Item / Phase	Prototype	Pre-release	Mature
exchange format (Unicode, W3C, OGN, NetCDF, etc)	X		
protocols (HTTP, TCP, TLS, etc)	X		

Checkmatrix for 'eStep friendly' projects.

Printable check-list -- complete this checklist to ensure your project is eStep-ready. If you can tick all boxes on this form, your project should be included as an eStep *Prototype* project.

Version Control

Item / Phase	Done
use git as version control system (vcs)	
public vcs repository (github)	
meaningful commit messages	

Licensing

Item / Phase	Done

Item / Phase	Done
Apache 2 license	
compatible license of all libraries	
`NOTICE(.txt or .md)` listing licenses, request citation of paper if applicable	

Communication

Item / Phase	Done
project discussion list (github issues, mailing list, not private email) for all project related discussions from the beginning of the project	

Documentation

Item / Phase	Done
`README.md` - clear explanation of the goal of the project with pointers to other documentation resources. Use **GitHub flavored markdown** for, e.g., **syntax highlighting**.	

Use standards

Item / Phase	Done
exchange format (Unicode, W3C, OGN, NetCDF, etc)	
protocols (HTTP, TCP, TLS, etc)	

Checkmatrix for 'eStep friendly' projects.

Printable check-list -- complete this checklist to ensure your project is eStep-ready. If you can tick all boxes on this form, your project should be included as an eStep *Pre-release* project.

##Version Control

Item / Phase	Done
use git as version control system (vcs)	
use **GitHub flow branching model** (use feature branches and pull requests)	
public vcs repository (**github**)	
meaningful commit messages	

##Releases

Item / Phase	Done
discuss release cycle with coordinator	

##Licensing

Item / Phase	Done
Apache 2 license	
compatible license of all libraries	
`NOTICE(.txt or .md)` listing licenses, request citation of paper if applicable	

##Communication

Item / Phase	Done
project discussion list (github issues, mailing list, not private email) for all project related discussions from the beginning of the project	
Pitch presentation (1 to 3 slides)	

##Testing

Item / Phase	Done
unit tests	
build tests	
continuous integration, public on Travis	

##Documentation

Item / Phase	Done
`README.md` - clear explanation of the goal of the project with pointers to other documentation resources. Use GitHub flavored markdown for, e.g., syntax highlighting.	
well defined functionality	
source code documentation	
usage documentation	
documented development setup (good example is Getting started with khmer development)	
contribution guidelines egzample	
code of conduct (contributor covenant)	
documented code style	
meaning of issue labels used	

Development setup

Item / Phase	Done
using the eScience Center coding style is required	
editorconfig	
applied code style in automated way if possible (i.e using linters and code formaters)	
dev environment docker images in Dockerhub (with Dockerfile)	

Use standards

Item / Phase	Done
exchange format (Unicode, W3C, OGN, NetCDF, etc)	
protocols (HTTP, TCP, TLS, etc)	

Checkmatrix for 'eStep friendly' projects.

Printable check-list -- complete this checklist to ensure your project is eStep-ready. If you can tick all boxes on this form, your project should be included as an eStep *Mature* project.

Version Control

Item / Phase	Done
use git as version control system (vcs)	
use GitHub flow branching model (use feature branches and pull requests)	
public vcs repository (github)	
meaningful commit messages	

Releases

Item / Phase	Done
semantic versioning	
tagged releases (github releases)	
CHANGELOG.md (Keep a CHANGELOG)	
one command install (pip, npm etc)	
package in package manager (pypi, npm etc)	
discuss release cycle with coordinator	

Item / Phase	Done
release quick-scan by other engineer (is documentation understandable, can it be installed, etc)	
notify Lode for dissemination (news item on site / annual report, etc)	

Licensing

Item / Phase	Done
Apache 2 license	
compatible license of all libraries	
`NOTICE(.txt or .md)` listing licenses, request citation of paper if applicable	

Communication

Item / Phase	Done
home page with all the necessary introduction information, links to documenation, source code (github) and latest release download (eg. **github.io pages**)	
project discussion list (github issues, mailing list, not private email) for all project related discussions from the beginning of the project	
for services: a demo docker image in dockerhub (with Dockerfile)	
for websites: an online demo	
Pitch presentation (1 to 3 slides)	
Few sentences about the project for **the technology pages on our website**	

Testing

Item / Phase	Done
unit tests	
build tests	
continuous integration, public on **Travis**	
continuous code coverage and code quality metrics public, minimum 70% coverage required	
end2end test for (web) user interfaces	
track dependencies (with **David** or other service depending on codebase language)	

Documentation

Item / Phase	Done
`README.md` - clear explanation of the goal of the project with pointers to other documentation resources. Use **GitHub flavored markdown** for, e.g., **syntax highlighting**.	
well defined functionality	
source code documentation	
usage documentation	
documented development setup (good example is **Getting started with khmer development**)	
contribution guidelines **egzample**	
code of conduct (**contributor covenant**)	
documented code style	
meaning of issue labels used	
DOI or PID (**making your code citable**)	

Development setup

Item / Phase	Done
using the eScienc Center coding style is required	
editorconfig	
applied code style in automated way if possible (i.e using linters and code formaters)	
dev environment docker images in Dockerhub (with Dockerfile)	

Use standards

Item / Phase	Done
exchange format (Unicode, W3C, OGN, NetCDF, etc)	
protocols (HTTP, TCP, TLS, etc)	