Dalton Developer's Guide

Release 0.0

Dalton developers

Oct 10, 2018

Contents

CHAPTER 1

Quick start

1.1 Clone and build the latest code

Clone the repository (master branch):

```
$ git clone --recursive https://gitlab.com/dalton/dalton.git
```

In case of LSDalton:

```
$ git clone --recursive https://gitlab.com/dalton/lsdalton.git
```

Build the code:

```
$ ./setup [--help]
$ cd build
$ make [-j4]
```

Run the test set:

```
$ ctest [-j4]
```

1.2 Want to contribute changes?

Fork either https://gitlab.com/dalton/dalton or https://gitlab.com/dalton/lsdalton, commit your changes, and file a merge request. Always use feature branches, never push to the master branch.

1.3 Developer mailing list

All developers should sign up to the dalton-wizards mailing list. Here you get information about Dalton meetings, get updates on general disussions, and get information about release re-scheduling and decision making outside the Dalton

meetings. To sign up contact either Hans Jørgen Jensen (hjj@sdu.dk), Trygve Helgaker (t.u.helgaker@kjemi.uio.no) or Kenneth Ruud (kenneth.ruud@uit.no).

CHAPTER 2

I have cloned before August 27 - what now?

2.1 What happened on August 27?

We have split each of the Dalton and LSDalton repositories into two, a public repository and a private repository. So instead of two repositories we now have four: two public (containing the `master` and release branches), and two private branches (containing all other branches):

```
https://gitlab.com/dalton/dalton/
https://gitlab.com/dalton/dalton-private/
https://gitlab.com/dalton/lsdalton/
https://gitlab.com/dalton/lsdalton-private/
```

Few changes may surprise you:

- You will not be able to push to the public repository directly - changes are integrated via merge requests from forks.
- You will not find any `master` branch or release branches on the private repository.
- A fork is like a copy of an entire repository but your fork will not automatically update itself, you have to keep it in sync.

2.2 If you have uncommitted or unpushed changes

You cannot simply push your changes because 1) you cannot push directly to the central repository and 2) the branch you want to push to is most probably not there anymore.

If you have changes on your local `master` branch, we recommend to create a feature branch from the `master` branch.

You have now two options to push your feature branches:

To the private repository (the “old” repository):

```
$ git push -u git@gitlab.com:dalton/dalton-private.git my-feature-branch
```

Or to a fork (adapt "user"). You may need to create the fork first:

```
$ git push -u git@gitlab.com:user/dalton.git my-feature-branch
```

Later you can submit a merge request from your fork. We recommend to push all modified branches one by one. After you are done, we recommend to create a fresh clone from your fork. Verify that your changes are there.

2.3 If you have no uncommitted or unpushed changes

If you are not sure whether you have uncommitted or unpushed changes on an old fork, please ask. If you are sure that you have no uncommitted or unpushed changes then we recommend to create a fresh clone, either from the central repository if you do not plan to make changes to the code:

```
$ git clone --recursive git@gitlab.com:dalton/dalton.git
$ git clone --recursive git@gitlab.com:dalton/lsdalton.git
```

Or if you wish to make changes to the code, first fork the repository and clone the fork (you need to adapt "user"):

```
$ git clone --recursive git@gitlab.com:user/dalton.git
$ git clone --recursive git@gitlab.com:user/lsdalton.git
```

CHAPTER 3

Working with branches and forks

3.1 Where are the branches?

We have split each of the Dalton and LSDalton repositories into two, a public repository and a private repository. So instead of two repositories we now have four: two public (containing the master and release branches), and two private repositories (containing all other branches):

```
https://gitlab.com/dalton/dalton/
https://gitlab.com/dalton/dalton-private/
https://gitlab.com/dalton/lsdalton/
https://gitlab.com/dalton/lsdalton-private/
```

3.2 The master branch

The `master` branches on the public repositories are the main development lines. Features should be developed on separate feature branches on the private repositories. The semantics of the `master` branch is that everything that is committed or merged to `master` is ready to be released in an upcoming release.

3.3 Release branches

Release branches are located on the public repositories, and are labeled `release/2013`, `release/2015`, etc. They branch from `master` once at feature freeze. They collect patches. All commits to release branches can be merged to `master` in principle at any moment. We never merge branches to release branches (except when accepting merge requests).

`release/2013` branch becomes frozen once Dalton 2015 is released. `release/2015` branch becomes frozen once Dalton 2016 is released. And so on.

3.4 How to make changes to master or the release branches

Never push changes directly to https://gitlab.com/dalton/dalton/ or https://gitlab.com/dalton/lsdalton/. Instead do this:

1. Create a fork.
2. Clone the fork.
3. Create a feature branch. **Never commit any changes to master or a release branch**.
4. Commit to the feature branch.
5. Push the feature branch to the fork.
6. Submit a merge request. You can mark the feature branch to be automatically deleted once the merge request is accepted. make sure that you do not forget any of the following points:

- Do not submit unfinished work.
- Be aware that everything on `master` will be part of the next major release.
- Describe your change in the CHANGELOG.
- Commits should have the right size (not too small, not too large) - otherwise squash or split changes.
- Commit messages should be clear.
- Commits should be self contained
- Test new code.
- Comment new code.
- Document new code.
- When fixing an issue, autoclose the issue with the commit message.

All commits to the master branch should be self-contained, eg. fix a bug or add a new feature, the commit message should clearly state what is achieved by the commit, and each commit should compile and run through the test suite. Long-term developments typically have several commits, many of which break compilation or test cases, and merging such developments directly will both make the git history ugly, and break functionality like ‘‘git bisect’‘. Such developments should therefore be squashed into one or a few commits.

3.5 Squashing

Assume you are working with two remote repositories “origin” and “upstream” (see how below). You have developed a new feature branch “my_messy_branch” on “origin” and incorporated the changes from “upstream” “master” into it, and would like to merge your changes back to “upstream” “master”.

First, make a local copy of your branch “my_messy_branch”:

```
$ git branch -b my_clean_branch
```

Second, rebase ‘‘my_clean_branch” with respect to “upstream” “master”:

```
$ git rebase upstream master
```

Third, make a ‘‘soft” ‘‘reset’‘

$ git reset –soft upstream master

This step removes all local commits, but leaves the code unchanged. This can be verified with a ‘‘git status’‘.

Fourth, make a commit of all the changes from “my_messy_branch” into one single commit on “my_clean_branch”, by a regular ‘‘git commit’‘, followed by a ‘‘git push origin my_clean_branch’‘, and a subsequent merge request on gitlab.

3.6 How to submit a merge request

https://docs.gitlab.com/ee/gitlab-basics/add-merge-request.html

3.7 Where to keep feature branches

You can either develop them on one of the private repositories (discouraged), or on your fork (encouraged). The fork can be either private or public.

3.8 How to update feature branches

You have probably cloned from your fork (good!) but from time to time you wish to update your feature branch(es) with changes on `master`.

First make sure that your clone is really cloned from your fork:

```
$ git remote -v
```

The remote `origin` should now point to your fork - here is an example:

```
origin git@gitlab.com:user/dalton.git (fetch)
origin git@gitlab.com:user/dalton.git (push)
```

We encourage you to regard the `master` branch as read-only and discourage you from making any changes to the `master` branch directly.

To update your forked `master` branch and your feature branches first add the central repository as an additional remote. You can call it for instance “upstream”:

```
$ git remote add upstream git@gitlab.com:dalton/dalton.git
```

Verify that it worked:

```
$ git remote -v

origin git@gitlab.com:user/dalton.git (fetch)
origin git@gitlab.com:user/dalton.git (push)
upstream git@gitlab.com:dalton/dalton.git (fetch)
upstream git@gitlab.com:dalton/dalton.git (push)
```

Good! Now you can fetch changes from upstream and merge master to your feature branch:

```
$ git fetch upstream
$ git checkout my-feature-branch
$ git merge upstream/master
$ git push origin my-feature-branch
```

3.9 How to update your fork

A fork is a clone. It will not update itself. To update your fork you need to pull changes from the upstream repository and push them to your fork.

You can do this either by adding an alias to a remote:

```
$ git remote add upstream git@gitlab.com:dalton/dalton.git
$ git fetch upstream
$ git checkout master
$ git merge upstream/master
$ git push origin master
```

Once you realize that "origin" and "upstream" are nothing more than aliases to web URIs you can fetch and push changes directly without defining new remotes:

```
$ git pull git@gitlab.com:dalton/dalton.git master
$ git push git@gitlab.com:user/dalton.git master
```

For other branches it works exactly the same way. For Git, all branches are equivalent. It is only convention to attach `master` a special meaning.

CHAPTER 4

F.A.Q.

4.1 CMake Error: The source directory X does not appear to contain CMakeLists.txt

You have probably cloned without `--recursive`. To remedy this you do not have to clone again, simply run:

```
$ git submodule update --init --recursive
```

4.2 How do I know whether I need to run git submodule update after I pull or switch branches?

To see this run `git status`. If you then see external modules which you haven't touched as modified, then you can update them using:

```
$ git submodule update --init --recursive
```

Note that switching branches or pulling updates does not automatically update the submodules.

4.3 I have some problem with gitlab.com. Is this my problem or a problem on gitlab.com side?

To see whether gitlab.com has some known issue, please check:

- http://status.gitlab.com
- https://twitter.com/gitlabstatus
- https://gitlab.com/gitlab-org/gitlab-ce/issues

4.4 Do I need to create a ssh keypair on each and every machine I want to use Dalton?

No. You do not need to authenticate via ssh. You can clone via https:

```
$ git clone https://gitlab.com/dalton/dalton.git
```

It will ask you for your gitlab.com user and password. This is useful when you need to clone to a machine just once in a while (cluster) and where you don't do heavy development work.

CHAPTER 5

Working with Git submodules

The Dalton repository includes other repositories via Git submodules (which, in turn, may include other repositories).

Therefore we recommend to clone the repository with the `--recursive` flag:

```
$ git clone --recursive https://gitlab.com/dalton/dalton.git
```

When switching Dalton branches remember to update the references:

```
$ git submodule update --init --recursive
```

CHAPTER 6

Coding standards

6.1 Fortran90

- Always use “implicit none” when writing new code
- Always make your module “private”
- Seperate declaration of input variables with local variables
- Always put intent(in), intent(out), or intent(inout) on all arguments. Note these attributes are used by Doxygen to create meaningful documentation.
- If you use Fortran2008 features beware that not all compilers may support this
- When using optional arguments always use “present” inquire function before using it
- Be careful when using intent(out) for pointers and derived types (e.g. type(matrix)); this will on some platforms make the pointer disassociated entering the routine, and memory already allocated for the matrix will be lost. Instead, use intent(inout) for derived types which should really be intent(out).
- Never initialize a local variable when declaring it. A local variable that is initialized when declared has an implicit save attribute
- When you change your already documented code, REMEMBER TO CHANGE ALSO THE DOCUMENTATION! Otherwise the documentation will quickly become useless.
- Minimize dependencies between different program components.
- All variables/data structures/objects should be declared in the smallest scope possible. For example, if a variable can be made local to a subroutine, rather than coming as input from a higher-level subroutine, this is preferable. Sometimes time-critical or memory-critical subroutines/functions may need to reduce the number of local variables. In such cases this should be clearly commented.
- The behavior of a subroutine/function should depend only on its input arguments. Access to variables that are neither subroutine arguments nor local variables inside a subroutine (e.g. global variables in F77 common blocks or Fortran 90 modules) introduce global data-dependencies which by definition break the modularity and make it harder to ensure the correctness of the program.

- Subroutines/functions should receive as little extraneous data as possible. That is, the input arguments should not contain more that than is necessary to produce the desired output.
- When it is easy (and non-time consuming) to do so, subroutines/functions should check that its input is reasonable. Even when a logically airtight test is impractically complicated, it is typically simple to test that the input satisfies some conservative preconditions. When the input data is unreasonable, this should be flagged in the output somehow and checked by the calling subroutine/function.
- Data structures should ideally represent concepts that are natural from the point of view of problem domain and/or algorithm at hand, and reflect the terms in which the programmer thinks about the functionality.
- Adopting some principles of Object-Oriented Programming is a good idea. For example, Fortran 90 supports some encapsulation and data-hiding through the keyword PRIVATE. Use of PRIVATE whereever reasonable is encouraged as it reduces bugs as well as makes it harder for other programmers to deviate from the intended usage of data structures and/or subroutines.
- The ideal structure of a program or directory or module is a library-like hierarchy of subroutines/functions/objects. At the lowest level in the hierarchy are "level 0" subroutines/functions/objects that are either self-contained or depend only on each other. At the next level are "level 1" subroutines/functions/objects that depend on each other and "level 0" subroutines/functions/objects. "Level 2" subroutines/functions/objects should ideally depend on each other and "level 1" subroutines/functions/objects, and not on "level 0", and so on. Document what the purpose and intended usage is of the different levels.
- A subroutine/function should not be longer than approximately one A4 page. Long subroutines/functions should be broken down into smaller components.
- Duplicate functionality as little as possible, within the constraints of producing readable, modular code. Functionality that is needed in several places should be implemented in a reusable, library-like way.

6.2 LSDalton

- Exit Document:

  People leaving the group should write an "exit document" which should contain some comments about the code - some general info about - the main driver - the general scheme - general thinking - basic idea - so that a new developer would have some idea to know where to start modifing the code.

- Print Level:

  The LSDALTON.OUT file contains way too much information compared to what the "user" need. We suggest to start reducing printout significantly.

  The default printlevel should be 0 which only prints the very basics like,

  Input files (full) possible DFT grid info SCF convergence Final SCF energy CCSD convergence Final CCSD energy Time for full calculation

  basicly nothing else.

- Restart files:

  1) All restart files should be names *.restart - currently we have files like overlapmatrix and cmo_orbitals.u and DECorbitals.info

  2) It would be nice that the lcm_orbitals.u (renamed to lcm_orbitals.restart) would be the final orbitals if the localization converged and the file from which the localization could be restarted from if it did not.

  3) The keyword ".RESTART" is used under *DENSOPT, *CC, **DEC but under **LOCALIZE ORBITALS it is called ".RESTART LOCALIZATION" which should be changed.

- User forbidden features:

  The global logical "developer" in ls_parameters.F90 is an attempt to avoid that the general users tries out bad keyword combinations.

  For example one method that might be available for full molecular calculations (**CC) but that has not been properly tested with **DEC, you might one to prevent the general user to combine those keywords by using the "developer" globale logical variable. It is false by default. But you can make true e.g. by setting:

```
**GENERAL
.DEVELOPER
```

- Divide Expand Consolidate (deccc directory)

  1. New DEC model/correction:

     If you want to implement energies for a new model/correction in DEC, then please do the following:

     a) Define whether you want to introduce a new MODEL (e.g. CC3) OR a new CORRECTION (e.g. F12) which can be added to various existing models/corrections.

     b) The global integers MODEL_* in dec_typedef.F90 define the various models. If you include a new model, then add it here. If you include a new correction (e.g. F12), then there is nothing to add here.

     c) Since we have different partitioning schemes for each CC model, it is necessary to have another set of global integers which define the fragment model (e.g. occupied partitioning scheme for CCSD). These are defined as the global integers FRAGMODEL_* in dec_typedef.F90. If you include a new model OR correction, then add it here. At the same time, you need to increase the global integer ndecenergies in lsutil/dec_typedef.F90 accordingly. E.g. if your new model/correction requires N additional energies to be stored, increase ndecenergies by N.

     (d) New model:

        i. Define your model in dec_set_model_names and find_model_number_from_input.

        ii. Add model to input keyword(s) in config_dec_input. (For example, see what is done for CCSD and do something similar).

     New correction:

        iii. Insert logical describing whether correction should be done in type DECsettings in lsutil/dec_typedef.f90.

        iv. Set defaults in dec_set_default_config and add input keyword(s) in config_dec_input. (For example, see what is done for F12 and do something similar).

  5. For new model (e.g. RPA): Grep for "MODIFY FOR NEW MODEL" in all files in the DEC code. These are the main places you need to change to include your new model. It is hopefully clear from the context and the comments in the code what needs to be done to include the new model.

     For new correction (e.g. F12): Grep for "MODIFY FOR NEW CORRECTION" in all files in the DEC code. These are the main places you need to change to include your new model. It is hopefully clear from the context and the comments in the code what needs to be to include the new correction".

  6. Workarounds:

     If you have to introduce workarounds on specific systems introduce them in dec_workarounds.F90 with a specific precompiler flag for your issue at hand. Rationale: Usually workarounds are compiler and system dependent and a clean version of the code should be maintained by default

Note: Feel free to update and improve this notes!

2. DEC dependencies:

   AGAIN, ALL MODULE SHOULD BE PRIVATE!

   The list below should ALWAYS be kept up to date!

   Rules:

   (a) Files at level X is allowed to use a subroutine/function in a file on level Y if and only if X>Y!

   (b) You are of course allowed to introduce new files, shuffle things around etc. If you do so, make sure that each files has a place in the hierarchy and that this readme-file is updated!

   (c) If modifying dependencies, make sure that all DEC depencies are put under the "! DEC DEPENDENCIES (within deccc directory)" comment within each file.

   Dependency list:

   Level 12 dec_main

   Level 11 snoop_main

   Level 10 full_driver, dec_driver, decmpiSlave, full_rimp2f12

   Level 9 dec_driver_slave

   Level 8 fragment_energy

   Level 7 cc_driver, cc_debug_routines

   Level 6 rpa,snoop_tools

   Level 5 pno_ccsd, ccsd, dec_atom, ccsdpt, mp2_gradient, f12_integrals, rif12_integrals, cc_response_tools

   Level 4 full_mp3

   Level 3 full_mp2

   Level 2 fullmolecule, mp2, cc_integrals, crop_tools, rimp2, ri_util

   Level 1 ccorbital, ccarray2_simple, ccarray3_simple, ccarray4_simple, decmpi, dec_utils

   Level 0 dec_settings, full_driver_f12contractions, CABS, cc_tools, f12ri_util array3_memory, array4_memory, f12_routines, dec_workarounds, dec_tools

   when you add a file to the deccc directory make sure to update this list as well as the one in the dec_readme file. (Maybe only one list should be kept)

CHAPTER 7

Tests

7.1 Running tests

We currently have two to four coexisting test mechanisms:

- Traditional shell script tests using `TEST` scripts; these are being phased out
- Perl scripts; these has been phased out
- Runtest based scripts, see: http://runtest.readthedocs.org
- CTest

CTest wraps around runtest. Runtest is for individual tests. CTest ties them all together and produces a test runner. The preferred way to run tests is through CTest inside the build directory.

You can run all tests:

```
$ ctest
```

all tests in parallel:

```
$ ctest -jN
```

You can match test names:

```
$ ctest -R somename
```

or labels:

```
$ ctest -L essential
```

to see all options, type:

```
$ man ctest
```

To see all labels, browse `cmake/Tests(LS)DALTON.cmake`.

You can also run tests individually, for this execute the individual python `test` scripts and point it to the correct build directory (try `./test -h` to see all options).

Warning: We should here describe what tests we require to pass before pushing anything to master.

7.2 Writing tests

Have a look here: http://runtest.readthedocs.org.

And have a look at other test scripts for inspiration. You have there full Python freedom. The important part is the return code. Zero means success, non-zero means failure.

To make a new test you have to make a folder, which is the test name, and put a mol and a dal file in that folder. You also have to include a reference output in a sub-folder "result/". Then you can just copy a "test" file from one of the other test directories and modify it according to your needs (there is no for this).

You have to add your new test to `cmake/Tests(LS)DALTON.cmake`. Then do:

```
$ cmake ..
```

in your build directory. Finally you can run your new test with the command:

```
$ ctest -R my_new_test
```

Your test folder will not be cluttered with output files if you are running the test from the build directory.

You can also adjust the test and execute it directly in the test directory but then be careful to not commit generated files.

7.3 Nightly testing dashboard

We have two testing dashboards, https://testboard.org/cdash/?project=Dalton, and https://testboard.org/cdash/?project=LSDalton.

This is the place to inspect tests which are known to fail (ideally none).

By default CTest will report to https://testboard.org/cdash/?project=Dalton. You can change this by setting `CTEST_PROJECT_NAME` to "LSDALTON" (or some other dashboard):

```
$ export CTEST_PROJECT_NAME=LSDALTON
```

By default the build name that appears on the dashboard is set to:

```
"${CMAKE_SYSTEM_NAME}-${CMAKE_HOST_SYSTEM_PROCESSOR}-${CMAKE_Fortran_COMPILER_ID}-$
↪{BLAS_TYPE}-${CMAKE_BUILD_TYPE}"
```

If you don't like it you can either change the default, or set the build name explicitly:

```
$ ./setup -D BUILDNAME='a-better-build-name'
```

Then run CTest with -D Nightly or Experimental:

```
$ ctest -D Nightly      [-jN] [-L ...] [-R ...]
$ ctest -D Experimental [-jN] [-L ...] [-R ...]
```

If you want to test your current code, take Experimental. If you want to set up a cron script to run tests every night, take Nightly.

By default CTest will report to https://testboard.org/cdash/?project=Dalton. You can change this by setting `CTEST_PROJECT_NAME` to "LSDALTON" (or some other dashboard):

```
$ export CTEST_PROJECT_NAME=LSDALTON
```

On a mobile device you may find this useful: https://testboard.org/cdash/iphone/project.php?project=Dalton

7.4 Testing with coverage

To compile and test for coverage is to collect statistics on which source lines are executed by the test suite. This accomplished at the setup stage:

```
$ ./setup --type=debug --coverage
```

and by executing the tests with:

```
$ ctest -D Experimental
```

This will execute the tests and upload the statistics to the CDash, and listed under *Coverage*. For each source file one obtains the percentage of executable lines that were executed by the test suite or the selected tests. To be able to see which lines where executed and which were not executed, one has to be authorized and logged in to the CDash pages.

7.5 Viewing coverage results locally (GNU)

This section applies for the GNU compiler suite. It can be useful if one wants to check things locally without submitting each run to the public site.

- The setup is the same as above, the `--coverage` compiler option is a synonym for the compiler flags `-fprofile-arcs -ftest-coverage` and link flag `-lgov`.
- During the compile stage, for each source file `xxx.F` a file `xxx.F.gcno` is generated
- When the tests are run, for each source file `xxx.F` a file `xxx.F.gcda` is generated
- The `gcov` program is used to obtain text files with coverage results. This can be used for individual files: *e.g.*, in the build directory run the command:

```
$ gcov CMakeFiles/dalton.dir/DALTON/yyy/xxx.F.gcno
File '.../dalton/DALTON/yyy/xxx.F'
Lines executed:86.37% of 653
Creating 'xxx.F.gcov'
```

This generates a copy of the source file where each source line is preceded by exexecution count and line number. In particular, lines that have not been executed by the tests are labeled `#####`

- The `lcov` program is a graphical frontend and the following steps can be used to generate html locally:

```
$ lcov -o xxx.info -c -d CMakeFiles/dalton.dir/DALTON
$ genhtml -o result xxx.info
```

Open result/index.html in your browser and you have a local graphical view of coverage statistics down to the source-line level.

CHAPTER 8

Reporting and fixing bugs

8.1 Where to report bugs

We track issues at https://gitlab.com/dalton/dalton/issues. Please report and discuss bugs there.

In addition you can discuss serious bugs in the developers section of the Dalton forum.

8.2 How to fix bugs

Previous text was outdated and removed. If you know how to do it, please document here.

CHAPTER 9

How you can contribute to this documentation

These pages are rendered using RST/Sphinx and served using Read the Docs. RST is a subset of Sphinx. Sphinx is RST with some extensions.

9.1 How to modify the webpages

The source code for this documentation is hosted on https://gitlab.com/dalton/devguide/. You need a GitLab account to modify the sources.

With a GitLab account you have two possibilities to edit the sources:

- If you are member of the dalton group, you can push directly to https://gitlab.com/dalton/devguide/. Once you commit and push, a web-hook updates the documentation on http://dalton-devguide.readthedocs.io. This typically takes less than a minute.
- You fork https://gitlab.com/dalton/devguide/ and submit your changes at some point via a merge request. This means that your changes are not immediately visible but become so after a team member reviews your changes with a mouse click thus integrating them to https://gitlab.com/dalton/devguide/.

Note that the entire documentation including the entire documentation source code is public. Do not publish sensitive information and harvestable email addresses.

9.2 How to locally test changes

You do not have to push to see and test your changes. You can test them locally. For this install the necessary requirements:

```
$ virtualenv venv
$ source venv/bin/activate
$ pip install sphinx sphinx_rtd_theme
```

Then build the pages with:

```
$ sphinx-build . _build
```

Then point your browser to _build/html/index.html.