Autocmake Documentation

Release 0.5.0

Radovan Bast and Jonas Juselius

May 20, 2016

Contents

CHAPTER 1

General

1.1 About Autocmake

Building libraries and executables from sources can be a complex task. Several solutions exist to this problem: GNU Makefiles is the traditional approach. Today, CMake is one of the trendier alternatives which can generate Makefiles starting from a file called `CMakeLists.txt`. Autocmake composes CMake building blocks into a CMake project and generates `CMakeLists.txt` as well as a setup script, which serves as a front-end to `CMakeLists.txt`. All this is done based on a lightweight `autocmake.cfg` file:

```
python update.py --self
     |                                   |
     | fetches Autocmake                 |
     | infrastructure                    |
     | and updates the update.py script  |
     |                                   |
     v                           Developer maintaining
autocmake.cfg                        Autocmake
     |                                   |
     | python update.py ..               |
     |                                   |
     v                                   v
CMakeLists.txt (and setup front-end)
     |                                   |
     | python setup                      |
     | which invokes CMake               |
     v                             User of the code
Makefile (or something else)             |
     |                                   |
     | make                              |
     |                                   |
     v                                   v
Build/install/test targets
```

Our main motivation to create Autocmake as a CMake framework library and CMake module composer was to simplify CMake code transfer between codes. We got tired of manually diffing and copy-pasting boiler-plate CMake code and watching it diverge while maintaining the CMake infrastructure in a growing number of scientific projects which typically have very similar requirements:

- Fortran and/or C and/or C++ support
- Tuning of compiler flags
- Front-end script with good defaults

- Support for parallelization: MPI, OMP, CUDA
- Math libraries: BLAS, LAPACK

Our other motivation for Autocmake was to make it easier for developers who do not know CMake to provide a higher-level entry point to CMake.

Autocmake is a chance to provide a well documented and tested set of CMake plug-ins. With this we wish to give also users of codes the opportunity to introduce the occasional tweak without the need to dive deep into CMake documentation.

1.2 Requirements and dependencies

Autocmake update and test scripts require Python 2.7 or higher. We try to also support Python 3 (tested with Python 3.4). If the script fails with Python 3, consider this a bug and please file an issue.

The generated setup script runs with Python >= 2.6 (also tested with Python 3.4; probably also lower).

1.3 Getting help

Autocmake discussion forum: http://groups.google.com/group/autocmake

CHAPTER 2

For developers who use Autocmake

2.1 FAQ for developers

2.1.1 Autocmake does not do feature X - I really need feature X and a setup flag –X

The Autocmake developers have to be very conservative and only a very limited set of portable features of absolutely general interest become part of the Autocmake core or an Autocmake module. Autocmake developers are also busy.

Our recommendation is to not wait for the feature to be implemented: Implement it yourself. Here we show you how. Code your feature in a module (i.e. `my_feature.cmake`) and place the module under `cmake/custom/` (the directory name is just a suggestion, Autocmake does not enforce a directory naming):

```
cmake/custom/my_feature.cmake
```

And include this feature to the main `CMakeLists.txt` in `autocmake.cfg`:

```
[my_feature]
source: custom/my_feature.cmake
```

Now your code is included in the main `CMakeLists.txt`. Perhaps you also want a setup script flag to toggle the feature:

```
[my_feature]
source: custom/my_feature.cmake
docopt: --my-feature Enable my feature [default: False].
define: '-DENABLE_MY_FEATURE={0}'.format(arguments['--my-feature'])
```

Implement your ideas, test them, and share them. If your module is portable, good code quality, and of general interest, you can suggest it to be part of the standard set of modules or even a core feature.

2.1.2 How can I get a setup flag –X that toggles a CMake variable?

The following will add a `--something` flag which toggles the CMake variable `ENABLE_SOMETHING`:

```
[my_section]
docopt: --something Enable something [default: False].
define: '-DENABLE_SOMETHING={0}'.format(arguments['--something'])
```

2.1.3 Can I change the name of the setup script?

Yes you can do that in `autocmake.cfg`. Here we for instance change the name to “configure”:

```
[project]
name: myproject
min_cmake_version: 2.8
setup_script: configure
```

2.1.4 In CMake I can do feature X - can I do that also with Autocmake?

Yes. Autocmake is really just a simplistic script which helps to organize CMake code across projects. Everything that can be done in CMake can be realized in Autocmake.

2.1.5 Should I include and track also files generated by Autocmake in my repository?

Yes, you probably want to do that. Autocmake generates a number of files which in principle could be generated at configure- or build-time. However, you probably do not want the users of your code to run any Autocmake scripts like `update.py` to generate the files they need to build the project. The users of your code will run `setup` directly and expect everything to just work (TM).

2.1.6 The update.py script is overwriting my CMakeLists.txt and setup, isn't this bad?

No, it is not as bad as it first looks. It is a feature. Normally `CMakeLists.txt` and `setup` should not contain any explicit customization and therefore should not contain anything that could not be regenerated. In any case you should use version control so that you can inspect and compare changes introduced to `CMakeLists.txt` and `setup` and possibly revert them. See also the next remark.

2.1.7 But I need to manually edit and customize CMakeLists.txt and setup every time I run update.py!?

You typically never need to manually edit and customize `CMakeLists.txt` and `setup` directly. You can introduce customizations in `autocmake.cfg` which get assembled into the front-end scripts.

2.1.8 Where is a good place to list my sources and targets?

As mentioned above `CMakeLists.txt` is not a good place because this file is generated from `autocmake.cfg` and your modifications would become overwritten at some point. A good standard is to organize your sources under `src/` and to list your sources and targets in `src/CMakeLists.txt`. You can include the latter in `autocmake.cfg` using:

```
[src]
source: https://github.com/coderefinery/autocmake/raw/stable-0.x/modules/src.cmake
```

If you really don't like to do it this way, you can describe your sources and targets in a custom module in a local file and include it like this:

```
[my_sources]
source: custom/my_sources.cmake
```

2.1.9 How can I do some more sophisticated validation of setup flags?

Sometimes you need to do more sophisticated validation and post-processing of setup flags. This can be done by placing a module called `extensions.py` under `cmake/` (or wherever you have `autocmake.cfg`). This file should implement a function with the following signature:

```python
def postprocess_args(sys_argv, arguments):
    # sys_argv is the sys.argv from the setup script
    # arguments is the dictionary of arguments returned by docopt

    # do something here ...

    return arguments
```

In this function you can do any validation and post-processing you like. This function is run after the flags are parsed and before the `CMake` command is run.

2.2 Bootstrapping a new project

2.2.1 Bootstrapping Autocmake

Download the `update.py` and execute it with `--self` to fetch other infrastructure files which will be needed to build the project:

```
$ mkdir cmake  # does not have to be called "cmake" - take the name you prefer
$ cd cmake
$ wget https://github.com/coderefinery/autocmake/raw/stable-0.x/update.py
$ python update.py --self
```

On the MS Windows system, you can use the PowerShell wget-replacement:

```
$ Invoke-WebRequest https://github.com/coderefinery/autocmake/raw/stable-0.x/update.py -OutFile updat
```

This creates (or updates) the following files (an existing `autocmake.cfg` is not overwritten by the script):

```
cmake/
    autocmake.cfg      # edit this file
    update.py          # no need to edit
    lib/
        config.py      # no need to edit
        docopt/
            docopt.py  # no need to edit
```

Note that all other listed files are overwritten (use version control).

2.2.2 Generating the CMake infrastructure

Now customize `autocmake.cfg` to your needs (see *Configuring autocmake.cfg*) and then run the `update.py` script which creates `CMakeLists.txt` and a setup script in the target path:

```
$ python update.py ..
```

The script also downloads external CMake modules specified in `autocmake.cfg` to a directory called `downloaded/`:

```
cmake/
    autocmake.cfg      # edit this file
    update.py          # no need to edit
    lib/
        config.py      # no need to edit
        docopt/
            docopt.py  # no need to edit
        downloaded/    # contains CMake modules fetched from the web
```

2.2.3 Building the project

Now you have `CMakeLists.txt` and setup script in the project root and the project can be built:

```
$ cd ..
$ python setup [-h]
$ cd build
$ make
```

2.3 Configuring autocmake.cfg

The script `autocmake.cfg` is the high level place where you configure your project. Here is an example. We will discuss it in detail further below:

```
[project]
name: numgrid
min_cmake_version: 2.8

[fc]
source: https://github.com/coderefinery/autocmake/raw/stable-0.x/modules/fc.cmake

[cc]
source: https://github.com/coderefinery/autocmake/raw/stable-0.x/modules/cc.cmake

[cxx]
source: https://github.com/coderefinery/autocmake/raw/stable-0.x/modules/cxx.cmake

[flags]
source: https://github.com/coderefinery/autocmake/raw/stable-0.x/compilers/GNU.CXX.cmake
        https://github.com/coderefinery/autocmake/raw/stable-0.x/compilers/Intel.CXX.cmake

[rpath]
source: custom/rpath.cmake

[definitions]
source: https://github.com/coderefinery/autocmake/raw/stable-0.x/modules/definitions.cmake

[coverage]
source: https://github.com/coderefinery/autocmake/raw/stable-0.x/modules/code_coverage.cmake

[safeguards]
source: https://github.com/coderefinery/autocmake/raw/stable-0.x/modules/safeguards.cmake

[default_build_paths]
source: https://github.com/coderefinery/autocmake/raw/stable-0.x/modules/default_build_paths.cmake
```

```
[src]
source: https://github.com/coderefinery/autocmake/raw/stable-0.x/modules/src.cmake

[googletest]
source: https://github.com/coderefinery/autocmake/raw/stable-0.x/modules/googletest.cmake

[custom]
source: custom/api.cmake
        custom/test.cmake
```

2.3.1 Name and order of sections

We see that the configuration file has sections. The only section where the name matters is `[project]`:

```
[project]
name: numgrid
min_cmake_version: 2.8
```

This is where we define the project name (here "numgrid"). This section has to be there and it has to be called "project" (but it does not have to be on top).

The names of the other sections do not matter to Autocmake. You could name them like this:

```
[project]
name: numgrid
min_cmake_version: 2.8

[one]
source: https://github.com/coderefinery/autocmake/raw/stable-0.x/modules/fc.cmake

[two]
source: https://github.com/coderefinery/autocmake/raw/stable-0.x/modules/cc.cmake

[whatever]
source: https://github.com/coderefinery/autocmake/raw/stable-0.x/modules/cxx.cmake
```

But it would not make much sense. It is better to choose names that are meaningful to you.

The order of the sections does matter and the sections will be processed in the exact order as you specify them in `autocmake.cfg`.

2.3.2 Minimal example

As a minimal example we take an `autocmake.cfg` which only contains:

```
[project]
name: minime
min_cmake_version: 2.8
```

First we make sure that the `update.py` script is up-to-date and that it has access to all libraries it needs:

```
$ python update.py --self

- creating .gitignore
- fetching lib/config.py
- fetching lib/docopt/docopt.py
- fetching update.py
```

Good. Now we can generate `CMakeLists.txt` and the setup script:

```
$ python update ..

- parsing autocmake.cfg
- generating CMakeLists.txt
- generating setup script
```

Excellent. Here is the generated `CMakeLists.txt`:

```
# set minimum cmake version
cmake_minimum_required(VERSION 2.8 FATAL_ERROR)

# project name
project(minime)

# do not rebuild if rules (compiler flags) change
set(CMAKE_SKIP_RULE_DEPENDENCY TRUE)

# if CMAKE_BUILD_TYPE undefined, we set it to Debug
if(NOT CMAKE_BUILD_TYPE)
    set(CMAKE_BUILD_TYPE "Debug")
endif()

set(CMAKE_MODULE_PATH ${CMAKE_MODULE_PATH} ${PROJECT_SOURCE_DIR}/cmake/downloaded)
```

This is the very bare minimum. Every Autocmake project will have at least these settings.

And we also got a setup script (front-end to `CMakeLists.txt`) with the following default options:

```
Usage:
  ./setup [options] [<builddir>]
  ./setup (-h | --help)

Options:
  --type=<TYPE>                          Set the CMake build type (debug, release, or relwithdeb) [de
  --generator=<STRING>                   Set the CMake build system generator [default: Unix Makefile
  --show                                 Show CMake command and exit.
  --cmake-executable=<CMAKE_EXECUTABLE>  Set the CMake executable [default: cmake].
  --cmake-options=<STRING>               Define options to CMake [default: ''].
  <builddir>                             Build directory.
  -h --help                              Show this screen.
```

That's not too bad although currently we cannot do much with this since there are no sources listed, no targets, hence nothing to build. We need to flesh out `CMakeLists.txt` by extending `autocmake.cfg` and this is what we will do in the next section.

2.3.3 Assembling CMake plugins

The preferred way to extend `CMakeLists.txt` is by editing `autocmake.cfg` and using the `source` option:

```
[fc]
source: https://github.com/coderefinery/autocmake/raw/stable-0.x/modules/fc.cmake
```

This will download `fc.cmake` and include it in `CMakeLists.txt`.

You can also include local CMake modules, e.g.:

```
[rpath]
source: custom/rpath.cmake
```

It is also OK to include several modules at once:

```
[flags]
source: https://github.com/coderefinery/autocmake/raw/stable-0.x/compilers/GNU.CXX.cmake
        https://github.com/coderefinery/autocmake/raw/stable-0.x/compilers/Intel.CXX.cmake
```

The modules will be included in the same order as they appear in `autocmake.cfg`.

2.3.4 Fetching files without including them in CMakeLists.txt

Sometimes you want to fetch a file without including it in `CMakeLists.txt`. This can be done with the `fetch` option. This is for instance done by the `git_info.cmake` module (see https://github.com/coderefinery/autocmake/blob/stable-0.x/modules/git_info/git_info.cmake#L10-L11).

If `fetch` is invoked in `autocmake.cfg`, then the fetched file is placed under `downloaded/`. If `fetch` is invoked from within a CMake module documentation (see below), then the fetched file is placed into the same directory as the CMake module file which fetches it.

2.3.5 Generating setup options

Options for the setup script can be generated with the `docopt` option. As an example, the following `autocmake.cfg` snippet will add a `--something` flag:

```
[my_section]
docopt: --something Enable something [default: False].
```

2.3.6 Setting CMake options

Configure-time CMake options can be generated with the `define` option. Consider the following example which toggles the CMake variable `ENABLE_SOMETHING`:

```
[my_section]
docopt: --something Enable something [default: False].
define: '-DENABLE_SOMETHING={0}'.format(arguments['--something'])
```

2.3.7 Setting environment variables

You can export environment variables at configure-time using the `export` option. Consider the following example:

```
[cc]
docopt: --cc=<CC> C compiler [default: gcc].
        --extra-cc-flags=<EXTRA_CFLAGS> Extra C compiler flags [default: ''].
export: 'CC=%s' % arguments['--cc']
define: '-DEXTRA_CFLAGS="%s"' % arguments['--extra-cc-flags']
```

2.3.8 Auto-generating configurations from the documentation

To avoid a boring re-typing of boilerplate `autocmake.cfg` code it is possible to auto-generate configurations from the documentation. This is the case for many core modules which come with own options once you have sourced them.

The lines following `# autocmake.cfg configuration::` are understood by the `update.py` script to infer `autocmake.cfg` code from the documentation. As an example consider https://github.com/coderefinery/autocmake/blob/stable-0.x/modules/cc.cmake#L20-L25. Here, `update.py` will infer the configurations for `docopt`, `export`, and `define`.

2.3.9 Overriding documented configurations

Configurable documented defaults can be achieved using interpolations. See for instance https://github.com/coderefinery/autocmake/blob/stable-0.x/modules/boost/boost.cmake#L33-L36. These can be modified within `autocmake.cfg` with a dictionary, e.g.: https://github.com/coderefinery/autocmake/blob/stable-0.x/test/boost_libs/cmake/autocmake.cfg#L9

2.4 Example Hello World project

This is a brief example for the busy and impatient programmer. For a longer tour please see *Configuring autocmake.cfg*.

We start with a mixed Fortran-C project with the following sources:

```
feature1.F90
feature2.c
main.F90
```

First thing we do is to create a directory `src/` and we move all sources there. This is not necessary for Autocmake but it is a generally good practice:

```
.
`-- src
    |-- feature1.F90
    |-- feature2.c
    `-- main.F90
```

Now we create `cmake/` and fetch `update.py`:

```
$ mkdir cmake
$ cd cmake/
$ wget https://raw.githubusercontent.com/coderefinery/autocmake/stable-0.x/update.py
$ python update.py --self
```

Now from top-level our file tree looks like this:

```
.
|-- cmake
|   |-- autocmake.cfg
|   |-- lib
|   |   |-- config.py
|   |   `-- docopt
|   |       `-- docopt.py
|   `-- update.py
`-- src
    |-- feature1.F90
    |-- feature2.c
    `-- main.F90
```

Now we edit `cmake/autocmake.cfg` to look like this:

```
[project]
name: hello
min_cmake_version: 2.8

[fc]
source: https://github.com/coderefinery/autocmake/raw/stable-0.x/modules/fc.cmake

[cc]
source: https://github.com/coderefinery/autocmake/raw/stable-0.x/modules/cc.cmake

[src]
source: https://github.com/coderefinery/autocmake/raw/stable-0.x/modules/src.cmake
```

What we have specified here is the project name and that we wish Fortran and C support. The `src.cmake` module tells CMake to include a `src/CMakeLists.txt`. We need to create `src/CMakeLists.txt` which can look like this:

```
add_executable(
    hello.x
    main.F90
    feature1.F90
    feature2.c
    )
```

We wrote that we want to get an executable "hello.x" built from our sources.

Now we have everything to generate `CMakeLists.txt` and a setup script:

```
$ cd cmake
$ python update ..
```

And this is what we got:

```
.
|-- CMakeLists.txt
|-- cmake
|   |-- autocmake.cfg
|   |-- downloaded
|   |   |-- autocmake_cc.cmake
|   |   |-- autocmake_fc.cmake
|   |   `-- autocmake_src.cmake
|   |-- lib
|   |   |-- config.py
|   |   `-- docopt
|   |       `-- docopt.py
|   `-- update.py
|-- setup
`-- src
    |-- CMakeLists.txt
    |-- feature1.F90
    |-- feature2.c
    `-- main.F90
```

Now we are ready to build:

```
$ python setup --fc=gfortran --cc=gcc

FC=gfortran CC=gcc cmake -DEXTRA_FCFLAGS="''" -DENABLE_FC_SUPPORT="ON" -DEXTRA_CFLAGS="''" -DCMAKE_BU

-- The C compiler identification is GNU 4.9.2
-- The CXX compiler identification is GNU 4.9.2
-- Check for working C compiler: /usr/bin/gcc
-- Check for working C compiler: /usr/bin/gcc -- works
-- Detecting C compiler ABI info
-- Detecting C compiler ABI info - done
-- Detecting C compile features
-- Detecting C compile features - done
-- Check for working CXX compiler: /usr/bin/c++
-- Check for working CXX compiler: /usr/bin/c++ -- works
-- Detecting CXX compiler ABI info
-- Detecting CXX compiler ABI info - done
-- Detecting CXX compile features
-- Detecting CXX compile features - done
-- The Fortran compiler identification is GNU 4.9.2
-- Check for working Fortran compiler: /usr/bin/gfortran
-- Check for working Fortran compiler: /usr/bin/gfortran  -- works
-- Detecting Fortran compiler ABI info
-- Detecting Fortran compiler ABI info - done
-- Checking whether /usr/bin/gfortran supports Fortran 90
-- Checking whether /usr/bin/gfortran supports Fortran 90 -- yes
-- Configuring done
-- Generating done
-- Build files have been written to: /home/user/example/build

   configure step is done
   now you need to compile the sources:
   $ cd build
   $ make

$ cd build/
$ make

Scanning dependencies of target hello.x
[ 25%] Building Fortran object src/CMakeFiles/hello.x.dir/main.F90.o
[ 50%] Building Fortran object src/CMakeFiles/hello.x.dir/feature1.F90.o
[ 75%] Building C object src/CMakeFiles/hello.x.dir/feature2.c.o
[100%] Linking Fortran executable hello.x
[100%] Built target hello.x
```

Excellent! But this was a lot of typing and file creating just to get a simple executable compiled!? Of course, all that could have been done with few command lines directly but now we have a cross-platform project and can extend it and customize it and we also got a front-end script and command-line parser for free. Now go out and explore more Autocmake modules and features.

2.5 Customizing CMake modules

The `update.py` script assembles modules listed in `autocmake.cfg` into `CMakeLists.txt`. Those that are fetched from the web are placed inside `downloaded/`. You have several options to customize downloaded CMake modules:

2.5.1 Directly inside the generated directory

The CMake modules can be customized directly inside `downloaded/` but this is the least elegant solution since the customizations may be overwritten by the `update.py` script (use version control).

2.5.2 Adapt local copies of CMake modules

A slightly better solution is to download the CMake modules that you wish you customize to a separate directory (e.g. `custom/`) and source the customized CMake modules in `autocmake.cfg`. Alternatively you can serve your custom modules from your own http server.

2.5.3 Fork and branch the CMake modules

You can fork and branch the mainline Autocmake development and include the branched customized versions. This will make it easier for you to stay up-to-date with upstream development.

2.5.4 Overriding defaults

Some modules use interpolations to set defaults, see for instance https://github.com/coderefinery/autocmake/blob/stable-0.x/modules/boost/boost.cmake#L33-L36. These can be modified within `autocmake.cfg`, e.g.: https://github.com/coderefinery/autocmake/blob/stable-0.x/test/boost_libs/cmake/autocmake.cfg#L9

2.5.5 Create own CMake modules

Of course you can also create own modules and source them in `autocmake.cfg`.

2.5.6 Contribute customizations to the "standard library"

If you think that your customization will be useful for other users as well, you may consider contributing the changes directly to https://github.com/coderefinery/autocmake/. We very much encourage such contributions. But we also strive for generality and portability.

2.6 Updating CMake modules

To update CMake modules fetched from the web you need to run the `update.py` script:

```
$ cd cmake
$ python update.py ..
```

The CMake modules are not fetched or updated at configure time or build time. In other words, if you never re-run `update.py` script and never modify the CMake module files, then the CMake modules will remain forever frozen.

2.6.1 How to pin CMake modules to a certain version

Sometimes you may want to avoid using the latest version of a CMake module and rather fetch an older version, for example with the hash `abcd123`. To achieve this, instead of:

```
[foo]
source: https://github.com/coderefinery/autocmake/raw/stable-0.x/modules/foo.cmake
```

pin the version to `abcd123` (you do not need to specify the full Git hash, a unique beginning will do):

```
[foo]
source: https://github.com/coderefinery/autocmake/raw/abcd123/modules/foo.cmake
```

CHAPTER 3

For users of projects which use Autocmake

3.1 FAQ for users

3.1.1 TL;DR How do I compile the code?

```
$ python setup [-h]
$ cd build
$ make
```

3.1.2 How can I specify the compiler?

By default the setup script will attempt GNU compilers. You can specify compilers manually like this:

```
$ python setup --fc=ifort --cc=icc --cxx=icpc
```

3.1.3 How can I add compiler flags?

You can do this with `--extra-fc-flags`, `--extra-cc-flags`, or `--extra-cxx-flags` (depending on the set of enabled languages):

```
$ python setup --fc=gfortran --extra-fc-flags='-some-exotic-flag'
```

3.1.4 How can I redefine compiler flags?

If you export compiler flags using the environment variables `FCFLAGS`, `CFLAGS`, or `CXXFLAGS`, respectively, then the configuration will use those flags and neither augment them, nor redefine them. Setting these environment variables you have full control over the flags without editing CMake files.

3.1.5 How can I select CMake options via the setup script?

Like this:

```
$ python setup --cmake-options='"-DTHIS_OPTION=ON -DTHAT_OPTION=OFF"'
```

We use two sets of quotes because the shell swallows one set of them before passing the arguments to Python. If you do not use two sets of quotes then the setup command may end up incorrectly saved in *build/setup_command*.

CHAPTER 4

For developers/contributors to Autocmake

4.1 Contributing guidelines

Please follow the excellent guide: http://www.contribution-guide.org.

We do not require any formal copyright assignment or contributor license agreement. Any contributions intentionally sent upstream are presumed to be offered under terms of the OSI-approved BSD 3-clause license.

4.2 Testing Autocmake

You will need to install pytest.

Check also the Travis build and test recipe for other requirements.

Your contributions and changes should preserve the test set. You can run locally all tests with:

```
$ py.test test/test.py
```

You can also select individual tests, for example those with `fc_blas` string in the name:

```
$ py.test -k fc_blas test/test.py
```

For more options, see the `py.test` flags.

This test set is run upon each push to the central repository. See also the Travis build and test history.

4.3 Contributing to the documentation

Contributions and patches to the documentation are most welcome.

This documentation is refreshed upon each push to the central repository.

The module reference documentation is generated from the module sources using `#.rst:` tags (compare for instance http://autocmake.readthedocs.org/en/latest/module-reference.html#cc-cmake with https://github.com/coderefinery/autocmake/blob/stable-0.x/modules/cc.cmake).

Please note that the lines following `# autocmake.cfg configuration::` are understood by the `update.py` script to infer autocmake.cfg code from the documentation. As an example consider https://github.com/coderefinery/autocmake/blob/stable-0.x/modules/cc.cmake#L20-L25. Here, `update.py` will infer the configurations for `docopt`, `export`, and `define`.

CHAPTER 5

Reference

5.1 Module reference

5.1.1 boost.cmake

[Source code]

Detect, build, and link Boost libraries. This modules downloads the .zip archive from SourceForge at Autocmake update time. Note that the build-up commands are not Windows-compatible!

Your autocmake.cfg should look like this:

```
[boost]
override: {'major': 1, 'minor': 59, 'patch': 0, 'components': 'chrono;timer;system'}
source: https://github.com/coderefinery/autocmake/raw/stable-0.x/modules/boost/boost.cmake
```

Cross-dependencies between required components are not checked for. For example, Boost.Timer depends on Boost.Chrono and Boost.System thus you should ask explicitly for all three. If the self-build of Boost components is triggered the *BUILD_CUSTOM_BOOST* variable is set to *TRUE*. The CMake target *custom_boost* is also added. You should use these two to ensure the right dependencies between your targets and the Boost headers/libraries, in case the self-build is triggered. For example:

```
if(BUILD_CUSTOM_BOOST)
  add_dependencies(your_target custom_boost)
endif()
```

will ensure that *your_target* is built after *custom_boost* if and only if the self-build of Boost took place. This is an important step to avoid race conditions when building on multiple processes.

Dependencies:

```
mpi                           - Only if the Boost.MPI library is a needed component
python_libs                   - Only if the Boost.Python library is a needed component
```

Variables used:

```
BOOST_MINIMUM_REQUIRED      - Minimum required version of Boost
BOOST_COMPONENTS_REQUIRED   - Components (compiled Boost libraries) required
PROJECT_SOURCE_DIR
PROJECT_BINARY_DIR
CMAKE_BUILD_TYPE
MPI_FOUND
BUILD_CUSTOM_BOOST
```

autocmake.cfg configuration:

```
major=1
minor=48
patch=0
components=''
fetch: https://github.com/coderefinery/autocmake/raw/stable-0.x/modules/boost/boost_unpack.cmake
       https://github.com/coderefinery/autocmake/raw/stable-0.x/modules/boost/boost_userconfig.cmake
       https://github.com/coderefinery/autocmake/raw/stable-0.x/modules/boost/boost_configure.cmake
       https://github.com/coderefinery/autocmake/raw/stable-0.x/modules/boost/boost_build.cmake
       https://github.com/coderefinery/autocmake/raw/stable-0.x/modules/boost/boost_install.cmake
       https://github.com/coderefinery/autocmake/raw/stable-0.x/modules/boost/boost_headers.cmake
       https://github.com/coderefinery/autocmake/raw/stable-0.x/modules/boost/boost_cleanup.cmake
       http://sourceforge.net/projects/boost/files/boost/%(major)s.%(minor)s.%(patch)s/boost_%(major)
docopt: --boost-headers=<BOOST_INCLUDEDIR> Include directories for Boost [default: ''].
        --boost-libraries=<BOOST_LIBRARYDIR> Library directories for Boost [default: ''].
        --build-boost=<FORCE_CUSTOM_BOOST> Deactivate Boost detection and build on-the-fly <ON/OFF>
define: '-DBOOST_INCLUDEDIR="{0}"'.format(arguments['--boost-headers'])
        '-DBOOST_LIBRARYDIR="{0}"'.format(arguments['--boost-libraries'])
        '-DFORCE_CUSTOM_BOOST="{0}"'.format(arguments['--build-boost'])
        '-DBOOST_MINIMUM_REQUIRED="%(major)s.%(minor)s.%(patch)s"'
        '-DBOOST_COMPONENTS_REQUIRED="%(components)s"'
```

5.1.2 cc.cmake

[Source code]

Adds C support. Appends EXTRA_CFLAGS to CMAKE_C_FLAGS. If environment variable CFLAGS is set, then the CFLAGS are used and no other flags are used or appended.

Variables used:

```
EXTRA_CFLAGS
```

Variables modified:

```
CMAKE_C_FLAGS
```

Environment variables used:

```
CFLAGS
```

autocmake.cfg configuration:

```
docopt: --cc=<CC> C compiler [default: gcc].
        --extra-cc-flags=<EXTRA_CFLAGS> Extra C compiler flags [default: ''].
export: 'CC={0}'.format(arguments['--cc'])
define: '-DEXTRA_CFLAGS="{0}"'.format(arguments['--extra-cc-flags'])
```

5.1.3 ccache.cmake

[Source code]

Adds ccache support. The user should export the appropriate environment variables to tweak the program's behaviour, as described in its manpage. Notice that some additional compiler flags might be needed in order to avoid unnecessary warnings.

Variables defined:

```
CCACHE_FOUND
```

autocmake.cfg configuration:

```
docopt: --ccache=<USE_CCACHE> Toggle use of ccache <ON/OFF> [default: ON].
define: '-DUSE_CCACHE="{0}"'.format(arguments['--ccache'])
```

5.1.4 code_coverage.cmake

[Source code]

Enables code coverage by appending corresponding compiler flags.

Variables modified (provided the corresponding language is enabled):

```
CMAKE_Fortran_FLAGS
CMAKE_C_FLAGS
CMAKE_CXX_FLAGS
```

autocmake.cfg configuration:

```
docopt: --coverage Enable code coverage [default: False].
define: '-DENABLE_CODE_COVERAGE=%s' % arguments['--coverage']
```

5.1.5 cxx.cmake

[Source code]

Adds C++ support. Appends EXTRA_CXXFLAGS to CMAKE_CXX_FLAGS. If environment variable CXXFLAGS is set, then the CXXFLAGS are used and no other flags are used or appended.

Variables used:

```
EXTRA_CXXFLAGS
```

Variables modified:

```
CMAKE_CXX_FLAGS
```

Environment variables used:

```
CXXFLAGS
```

autocmake.cfg configuration:

```
docopt: --cxx=<CXX> C++ compiler [default: g++].
        --extra-cxx-flags=<EXTRA_CXXFLAGS> Extra C++ compiler flags [default: ''].
export: 'CXX={0}'.format(arguments['--cxx'])
define: '-DEXTRA_CXXFLAGS="{0}"'.format(arguments['--extra-cxx-flags'])
```

5.1.6 default_build_paths.cmake

[Source code]

Sets binary and library output directories to ${PROJECT_BINARY_DIR}/bin and ${PROJECT_BINARY_DIR}/lib, respectively.

Variables modified:

```
CMAKE_RUNTIME_OUTPUT_DIRECTORY
CMAKE_LIBRARY_OUTPUT_DIRECTORY
```

5.1.7 definitions.cmake

[Source code]

Add preprocessor definitions (example: –add-definitions=”-DTHIS -DTHAT=137”). These are passed all the way down to the compiler.

Variables used:

```
PREPROCESSOR_DEFINITIONS
```

autocmake.cfg configuration:

```
docopt: --add-definitions=<STRING> Add preprocesor definitions [default: ''].
define: '-DPREPROCESSOR_DEFINITIONS="%s"' % arguments['--add-definitions']
```

5.1.8 fc.cmake

[Source code]

Adds Fortran support. Appends EXTRA_FCFLAGS to CMAKE_Fortran_FLAGS. If environment variable FCFLAGS is set, then the FCFLAGS are used and no other flags are used or appended.

Variables used:

```
EXTRA_FCFLAGS
```

Variables defined:

```
CMAKE_Fortran_MODULE_DIRECTORY
```

Variables modified:

```
CMAKE_Fortran_FLAGS
```

Environment variables used:

```
FCFLAGS
```

autocmake.cfg configuration:

```
docopt: --fc=<FC> Fortran compiler [default: gfortran].
        --extra-fc-flags=<EXTRA_FCFLAGS> Extra Fortran compiler flags [default: ''].
export: 'FC={0}'.format(arguments['--fc'])
define: '-DEXTRA_FCFLAGS="{0}"'.format(arguments['--extra-fc-flags'])
```

5.1.9 fc_optional.cmake

[Source code]

Adds optional Fortran support. Appends EXTRA_FCFLAGS to CMAKE_Fortran_FLAGS. If environment variable FCFLAGS is set, then the FCFLAGS are used and no other flags are used or appended.

Variables used:

```
EXTRA_FCFLAGS
ENABLE_FC_SUPPORT
```

Variables defined:

```
CMAKE_Fortran_MODULE_DIRECTORY
```

Variables modified:

```
CMAKE_Fortran_FLAGS
```

Variables defined:

```
ENABLE_FC_SUPPORT
```

Environment variables used:

```
FCFLAGS
```

autocmake.cfg configuration:

```
docopt: --fc=<FC> Fortran compiler [default: gfortran].
        --extra-fc-flags=<EXTRA_FCFLAGS> Extra Fortran compiler flags [default: ''].
        --fc-support=<FC_SUPPORT> Toggle Fortran language support (ON/OFF) [default: ON].
export: 'FC={0}'.format(arguments['--fc'])
define: '-DEXTRA_FCFLAGS="{0}"'.format(arguments['--extra-fc-flags'])
        '-DENABLE_FC_SUPPORT="{0}"'.format(arguments['--fc-support'])
```

5.1.10 git_info.cmake

[Source code]

Creates git_info.h in the build directory. This file can be included in sources to print Git repository version and status information to the program output.

autocmake.cfg configuration:

```
fetch: https://github.com/coderefinery/autocmake/raw/stable-0.x/modules/git_info/git_info_sub.cmake
       https://github.com/coderefinery/autocmake/raw/stable-0.x/modules/git_info/git_info.h.in
```

5.1.11 git_info_sub.cmake

[Source code]

Creates git_info.h in the build directory. This file can be included in sources to print Git status information to the program output for reproducibility reasons.

5.1.12 googletest.cmake

[Source code]

Includes Google Test sources and adds a library "googletest".

Variables used:

```
GOOGLETEST_ROOT
```

autocmake.cfg configuration:

```
define: '-DGOOGLETEST_ROOT=external/googletest/googletest'
```

5.1.13 int64.cmake

[Source code]

Enables 64-bit integer support for Fortran projects.

Variables modified (provided the corresponding language is enabled):

```
CMAKE_Fortran_FLAGS
```

autocmake.cfg configuration:

```
docopt: --int64 Enable 64bit integers [default: False].
define: '-DENABLE_64BIT_INTEGERS=%s' % arguments['--int64']
```

5.1.14 accelerate.cmake

[Source code]

Find and link to ACCELERATE.

Variables defined:

```
ACCELERATE_FOUND - describe me, uncached
ACCELERATE_LIBRARIES - describe me, uncached
ACCELERATE_INCLUDE_DIR - describe me, uncached
```

autocmake.cfg configuration:

```
docopt: --accelerate Find and link to ACCELERATE [default: False].
define: '-DENABLE_ACCELERATE=%s' % arguments['--accelerate']
fetch: https://github.com/coderefinery/autocmake/raw/stable-0.x/modules/find/find_libraries.cmake
       https://github.com/coderefinery/autocmake/raw/stable-0.x/modules/find/find_include_files.cmake
```

5.1.15 acml.cmake

[Source code]

Find and link to ACML.

Variables defined:

```
ACML_FOUND
ACML_LIBRARIES
ACML_INCLUDE_DIR
```

autocmake.cfg configuration:

```
docopt: --acml Find and link to ACML [default: False].
define: '-DENABLE_ACML=%s' % arguments['--acml']
```

5.1.16 atlas.cmake

[Source code]

Find and link to ATLAS.

Variables defined:

```
ATLAS_FOUND
ATLAS_LIBRARIES
ATLAS_INCLUDE_DIR
```

autocmake.cfg configuration:

```
docopt: --atlas Find and link to ATLAS [default: False].
define: '-DENABLE_ATLAS=%s' % arguments['--atlas']
```

5.1.17 blas.cmake

[Source code]

Find and link to BLAS.

Variables defined:

```
BLAS_FOUND
BLAS_LIBRARIES
BLAS_INCLUDE_DIR
```

autocmake.cfg configuration:

```
docopt: --blas Find and link to BLAS [default: False].
define: '-DENABLE_BLAS=%s' % arguments['--blas']
```

5.1.18 cblas.cmake

[Source code]

Find and link to CBLAS.

Variables defined:

```
CBLAS_FOUND - describe me, uncached
CBLAS_LIBRARIES - describe me, uncached
CBLAS_INCLUDE_DIR - describe me, uncached
```

autocmake.cfg configuration:

```
docopt: --cblas Find and link to CBLAS [default: False].
define: '-DENABLE_CBLAS=%s' % arguments['--cblas']
fetch: https://github.com/coderefinery/autocmake/raw/stable-0.x/modules/find/find_libraries.cmake
       https://github.com/coderefinery/autocmake/raw/stable-0.x/modules/find/find_include_files.cmake
```

5.1.19 goto.cmake

[Source code]

Find and link to Goto BLAS.

Variables defined:

```
GOTO_FOUND
GOTO_LIBRARIES
GOTO_INCLUDE_DIR
```

autocmake.cfg configuration:

```
docopt: --goto Find and link to GOTO [default: False].
define: '-DENABLE_GOTO=%s' % arguments['--goto']
```

5.1.20 lapack.cmake

[Source code]

Find and link to LAPACK.

Variables defined:

```
LAPACK_FOUND
LAPACK_LIBRARIES
LAPACK_INCLUDE_DIR
```

autocmake.cfg configuration:

```
docopt: --lapack Find and link to LAPACK [default: False].
define: '-DENABLE_LAPACK=%s' % arguments['--lapack']
```

5.1.21 lapacke.cmake

[Source code]

Find and link to LAPACKE.

Variables defined:

```
LAPACKE_FOUND - describe me, uncached
LAPACKE_LIBRARIES - describe me, uncached
LAPACKE_INCLUDE_DIR - describe me, uncached
```

autocmake.cfg configuration:

```
docopt: --lapacke Find and link to LAPACKE [default: False].
define: '-DENABLE_LAPACKE=%s' % arguments['--lapacke']
fetch: https://github.com/coderefinery/autocmake/raw/stable-0.x/modules/find/find_libraries.cmake
       https://github.com/coderefinery/autocmake/raw/stable-0.x/modules/find/find_include_files.cmak
```

5.1.22 math_libs.cmake

[Source code]

Detects and links to BLAS and LAPACK libraries.

Variables used:

```
MATH_LIB_SEARCH_ORDER, example: set(MATH_LIB_SEARCH_ORDER MKL ESSL OPENBLAS ATLAS ACML SYSTEM_NATIVE)
ENABLE_STATIC_LINKING
ENABLE_BLAS
ENABLE_LAPACK
BLAS_FOUND
LAPACK_FOUND
BLAS_LANG
LAPACK_LANG
BLAS_TYPE
LAPACK_TYPE
ENABLE_64BIT_INTEGERS
CMAKE_HOST_SYSTEM_PROCESSOR
BLACS_IMPLEMENTATION
MKL_FLAG
```

Variables defined:

```
MATH_LIBS
BLAS_FOUND
LAPACK_FOUND
```

Variables modified:

```
CMAKE_EXE_LINKER_FLAGS
```

Environment variables used:

```
MATH_ROOT
BLAS_ROOT
LAPACK_ROOT
MKL_ROOT
MKLROOT
```

autocmake.cfg configuration:

```
docopt: --blas=<BLAS> Detect and link BLAS library (auto or off) [default: auto].
        --lapack=<LAPACK> Detect and link LAPACK library (auto or off) [default: auto].
        --mkl=<MKL> Pass MKL flag to the Intel compiler and linker and skip BLAS/LAPACK detection (se
define: '-DENABLE_BLAS=%s' % arguments['--blas']
        '-DENABLE_LAPACK=%s' % arguments['--lapack']
        '-DMKL_FLAG=%s' % arguments['--mkl']
        '-DMATH_LIB_SEARCH_ORDER="MKL;ESSL;OPENBLAS;ATLAS;ACML;SYSTEM_NATIVE"'
        '-DBLAS_LANG=Fortran'
        '-DLAPACK_LANG=Fortran'
warning: 'This module is deprecated and will be removed in future versions'
```

5.1.23 mpi.cmake

[Source code]

Enables MPI support.

Variables used:

```
ENABLE_MPI
MPI_FOUND
```

Variables modified (provided the corresponding language is enabled):

```
CMAKE_Fortran_FLAGS
CMAKE_C_FLAGS
CMAKE_CXX_FLAGS
```

autocmake.cfg configuration:

```
docopt: --mpi Enable MPI parallelization [default: False].
define: '-DENABLE_MPI=%s' % arguments['--mpi']
```

5.1.24 omp.cmake

[Source code]

Enables OpenMP support.

Variables used:

```
ENABLE_OPENMP
OPENMP_FOUND
```

Variables modified (provided the corresponding language is enabled):

```
CMAKE_Fortran_FLAGS
CMAKE_C_FLAGS
CMAKE_CXX_FLAGS
```

autocmake.cfg configuration:

```
docopt: --omp Enable OpenMP parallelization [default: False].
define: '-DENABLE_OPENMP=%s' % arguments['--omp']
```

5.1.25 python_interpreter.cmake

[Source code]

Detects Python interpreter.

Variables used:

```
PYTHON_INTERPRETER         - User-set path to the Python interpreter
```

Variables defined:

```
PYTHONINTERP_FOUND         - Was the Python executable found
PYTHON_EXECUTABLE          - path to the Python interpreter
PYTHON_VERSION_STRING      - Python version found e.g. 2.5.2
PYTHON_VERSION_MAJOR       - Python major version found e.g. 2
PYTHON_VERSION_MINOR       - Python minor version found e.g. 5
PYTHON_VERSION_PATCH       - Python patch version found e.g. 2
```

autocmake.cfg configuration:

```
docopt: --python=<PYTHON_INTERPRETER> The Python interpreter (development version) to use. [default:
define: '-DPYTHON_INTERPRETER="%s"' % arguments['--python']
```

5.1.26 python_libs.cmake

[Source code]

Detects Python libraries and headers. Detection is done basically by hand as the proper CMake package will not find libraries and headers matching the interpreter version.

Dependencies:

```
python_interpreter          - Sets the Python interpreter for headers and libraries detection
```

Variables used:

```
PYTHONINTERP_FOUND          - Was the Python executable found
```

Variables defined:

```
PYTHONLIBS_FOUND            - have the Python libs been found
PYTHON_LIBRARIES            - path to the python library
PYTHON_INCLUDE_DIRS         - path to where Python.h is found
PYTHONLIBS_VERSION_STRING   - version of the Python libs found (since CMake 2.8.8)
```

5.1.27 safeguards.cmake

[Source code]

Provides safeguards against in-source builds and bad build types.

Variables used:

```
PROJECT_SOURCE_DIR
PROJECT_BINARY_DIR
CMAKE_BUILD_TYPE
```

5.1.28 save_flags.cmake

[Source code]

Take care of updating the cache for fresh configurations.

Variables modified (provided the corresponding language is enabled):

```
DEFAULT_Fortran_FLAGS_SET
DEFAULT_C_FLAGS_SET
DEFAULT_CXX_FLAGS_SET
```

5.1.29 src.cmake

[Source code]

Adds ${PROJECT_SOURCE_DIR}/src as subdirectory containing CMakeLists.txt.

5.1.30 version.cmake

[Source code]

Determine program version from file "VERSION" (example: "14.1") The reason why this information is stored in a file and not as CMake variable is that CMake-unaware programs can parse and use it (e.g. Sphinx). Also web-based hosting frontends such as GitLab automatically use the file "VERSION" if present.

Variables defined:

```
PROGRAM_VERSION
```