Core Cosmology Library Documentation

Release 2.X

LSST DESC

Oct 21, 2019

Getting Started

The Core Cosmology Library (CCL) is a standardized library of routines to calculate basic observables used in cosmology. It will be the standard analysis package used by the LSST Dark Energy Science Collaboration (DESC).

The core functions of this package include:

- Matter power spectra $P(k)$ from numerous models including CLASS, the Mira-Titan Emulator and halofit
- Hubble constant $H(z)$ as well as comoving distances $\chi(z)$ and distance moduli $\mu(z)$
- Growth of structure $D(z)$ and f
- Angular power spectra C_ℓ and correlation functions ξ for arbitrary combinations of tracers including galaxies, shear and number counts
- Halo mass function $\mathrm{d}n/\mathrm{d}M$ and halo bias $b(M)$
- Approximate baryonic modifications to the matter power spectra $\Delta^2_{\rm baryons}$
- Simple modified gravity extensions $\Delta f(z)$ and $\mu - \Sigma$

The source code is available on github at https://github.com/LSSTDESC/CCL.

CHAPTER 1

Installation

CCL can be installed from `pip`, `conda`, or directly from source. It is configured to install most of its requirements automatically. However, if you want to use CCL with Boltzmann codes like `CLASS` or `CAMB`, you will need to make sure the `Python` wrappers for these packages are installed as well. See the instructions for *Getting a Boltzmann Code* below.

Once installed, you can take CCL for a spin by following the *Quickstart* instructions.

1.1 With conda

Installing CCL with `conda` will also install a copy of `pycamb`, the `Python` wrapper for `CAMB` in your environment.

```
$ conda install -c conda-forge pyccl
```

1.2 With pip

The PyPi installation will actually build a new copy of CCL from source as the code is installed. In order to do this, you will need `CMake` installed on your system. See the instructions below for *Getting CMake*.

Once you have `CMake`, simply run:

```
$ pip install pyccl
```

1.3 Getting a Boltzmann Code

In order to use CCL with a Boltzmann code, you will need the `Python` wrappers for either `CLASS` or `CAMB`.

1.3.1 CLASS

In order to use CCL with `CLASS`, you need to install the `CLASS Python` wrapper `classy`. See their instructions here. Note that you may need to edit the `Makefile` in the `CLASS` repo to work with your system. Please report any installation issues to the `CLASS` developers on their issues tracker.

1.3.2 CAMB

If you are working in a `conda` environment, then `CAMB` is available via

```
$ conda install -c conda-forge camb
```

An installation with `pip` should work as well. See the CAMB repo for more details. Note that if you installed CCL with `conda`, `camb` should already be in your environment.

1.4 Getting CMake

The `pip` installation of CCL requires that `CMake` is installed on your system. `CMake` is available from package managers like `apt-get` and `homebrew`. You need version 3.2 or higher.

1.4.1 Ubuntu

```
$ sudo apt-get install cmake
```

1.4.2 OS X

On MacOS X you can either install with a DMG from this page or with a package manager such as homebrew, MacPorts, or Fink.

For instance with `homebrew`, you can run

```
$ brew install cmake
```

1.5 Known Installation Issues

1. For some Mac OSX versions, the standard `C` headers are not in the usual spot, resulting in an error of `fatal error: 'stdio.h' file not found`. This can be resolved with the command:

   ```
   $ sudo installer -pkg /Library/Developer/CommandLineTools/Packages/macOS_SDK_
   ↪headers_for_macOS_10.14.pkg -target /
   ```

   which will install all the required headers into `/usr/include`.

1.6 Uninstalling CCL

CCL can be uninstalled using the uninstallation functionality of the package manager (i.e., conda or pip) you used to install it. When in doubt, first try with conda and then try with pip. In either case, the command is

```
$ [conda|pip] uninstall pyccl
```

CHAPTER 2

Quickstart

CCL is structured around *Cosmology* objects which hold the cosmological parameters and any tabulated data associated with a given cosmology. The library then provides functions to compute specific quantities of interest. See the *pyccl* module and submodules for more details.

Further, CCL follows the following conventions:

- all units are non-h-inverse (e.g., Mpc as opposed to Mpc/h)
- the scale factor `a` is preferred over redshift `z`
- the *Cosmology* object always comes first in function calls
- argument ordering for power spectra is `(k, a)`

This example computes the comoving distance and HALOFIT non-linear power spectrum using the BBKS transfer function:

```
>>> import pyccl
>>> cosmo = pyccl.Cosmology(Omega_c=0.25, Omega_b=0.05,
                            h=0.7, n_s=0.95, sigma8=0.8,
                            transfer_function='bbks')
>>> pyccl.sigma8(cosmo)  # get sigma8
0.7999999999999998
>>> z = 1
>>> pyccl.comoving_radial_distance(cosmo, 1./(1+z))  # comoving distance to z=1 in Mpc
3303.5261651302458
>>> pyccl.nonlin_matter_power(cosmo, k=1, a=0.5)  # HALOFIT P(k) at k,z = 1,1
143.6828250598087
```

See *Notation, Models and Other Cosmological Conventions* for more details on the supported models for various cosmological quantities (e.g., the power spectrum) and the specification of the cosmological parameters.

CHAPTER 3

Citing CCL

This code has been released by DESC, although it is still under active development. It is accompanied by a journal paper that describes the development and validation of *CCL*, which you can find on the arxiv. If you make use of the ideas or software here, please cite that paper and provide a link to the repository `https://github.com/LSSTDESC/CCL`. You are welcome to re-use the code, which is open source and available under terms consistent with our LICENSE (BSD 3-Clause).

External contributors and DESC members wishing to use CCL for non-DESC projects should consult with the TJP working group conveners, ideally before the work has started, but definitely before any publication or posting of the work to the arXiv.

For free use of the `CLASS` library, the `CLASS` developers require that the `CLASS` paper be cited, CLASS II: Approximation schemes, D. Blas, J. Lesgourgues, T. Tram, arXiv:1104.2933, JCAP 1107 (2011) 034. The `CLASS` repository can be found in http://class-code.net.

The `CAMB` developers have released `CAMB` under the LGPL license with a few additional restrictions. Please read their LICENSE file for more information.

CHAPTER 4

Reporting Bugs / Contacting a Human

If you have encountered a bug in CCL report it to the Issues page. This includes problems during installation, running the tests, or while using the package. If possible, provide an example script that reproduces the error.

If you find a discrepancy between a computed quantity from CCL and another source (or from also from CCL) please describe the quantity you are computing. If possible, provide an example script and a plot demonstrating the inconsistency.

Finally, if all else fails, you can contact the administrators directly.

CHAPTER 5

Terms of Reference and Development Principles

This document lays out terms of reference for CCL developers and contributors in the wider TJP and LSST DESC context.

5.1 Background

CCL was created to provide theoretical predictions of large-scale structure statistics for DESC cosmological analyses but it has also become a publicly available tool.

5.2 Scope

CCL has a commitment to support key DESC analyses. In this regard, CCL aims to

- support cosmological models that are priorities for WGs (e.g., as defined by the "Beyond wCDM" taskforce);
- ensure sufficient flexibility such that modifications can be implemented at a later stage as needed (e.g., models for observational systematics or changes to how the non-linear power spectrum is computed for various parts of the analysis);
- support core predictions (e.g., shear correlation functions), but not necessarily every statistic that could be computed from those core predictions (e.g., COSEBIs).

CCL interface support is as follows.

- Public APIs are maintained in Python only (i.e., version bumping is not necessary following changes in the C interface).
- The C implementation can be used for speed as needed.
- Documentation should be maintained in the main README, readthedocs and the benchmarks folder.
- Examples will be maintained in a separate repository.

Boundaries of CCL (with respect to various models and systematics)

- Not all models need to have equal support. CCL aims to support the DESC analyses, not to support all possible statistics in all models.
- CCL should provide a generic framework to enable modeling of systematics effects. However, overly specific models for systematics, for example, for galaxy bias, detector effects or intrinsic alignments, are best located in other repositories (e.g., firecrown).

Boundaries of CCL (with respect to other TJP software)

- TJP will necessarily develop other software tools to support DESC cosmological analyses. CCL development should ensure internal consistency between the various tools by defining a common parameterization of theoretical quantities (e.g., kernels for Limber integration, the metric potentials, etc.).
- CCL development should also proceed in a manner to avoid significant duplication of effort within the collaboration.

5.3 Standards to establish the quality of code that can be part of CCL

- Contributions to CCL should be well-documented in python (only doc-string level docs are required, readthedocs does the rest). C code is expected to be commented, though not documented.
- Features added to CCL should pass required tests.
    1. one (publicly available) benchmark with an independent prediction should be provided to test the specific feature that has been added. The benchmark code can use CCL input as long as that is not affecting the validation procedure.
    2. Tests should run in continuous integration scheme used for CCL.
    3. Unit tests should be written following the suggestions of the PR reviewers.
- All PRs are code reviewed by at least one person (two are suggested for PRs that bring in new features or propose significant changes to the CCL API).
- CCL uses semantic versioning.
- We should aim to tag new versions with bug fixes on a monthly basis with the expectation that no release is made if no new bugs have been fixed.
- Critical bug fixes should be released almost immediately via bump of the micro version and a new release.

5.4 Guidelines towards contribution

- Contributions should be made via PRs, and such PRs should be reviewed as described above.
- Response to code review requests should be prompt and happen within a few days of being requested on github by the author of the PR. Code review by members of the CCL development team should happen within a week of the request being accepted.
- The CCL *Developer Guide* should be consulted by those making PRs.
- Developers should use standard DESC channels to interact with the team (e.g., Slack, telecons, etc.).
- Developers should seek consensus with the team as needed about new features being incorporated (especially for API changes or additions). This process should happen via the CCL telecons, slack channel, or github comments generally before a new feature is merged. More minor changes, like bug fixes or refactoring, do not necessarily need this process.

- External contributions. They come in different forms: bug fixes or new features. They should proceed in full awareness of the LSST DESC Publication Policy. In particular, the use of new features added to the library, but not officially released, may require more formal steps to be taken within the DESC. External contributors and DESC members wishing to use CCL for non-DESC projects should consult with the TJP working group conveners, ideally before the work has started, but definitely before any publication or posting of the work to the arXiv.

CHAPTER 6

API Changes and Stability

CCL follows the conventions of semantic versioning. Semantic versioning is a consistent set of rules for constructing CCL version numbers so that CCL users can understand when new releases have backwards compatible APIs.

An API breakage is anything that modifies the name or call signature of an existing public-facing function or variable in the `Python` code in such a way that the change is not backwards compatible. For example, removing a `Python` keyword argument is an API breakage. However, adding one is not. Significant changes to the behavior of a function also count as an API breakage (e.g. if an assumption is changed that causes different results to be produced by default). Fixing a bug that affects the output of a function does not count as an API breakage, unless the fix modifies function names and arguments too as described above.

If the API is broken, the CCL major version number must be incremented (e.g. from v1.5.0 to v2.0.0), and a new tagged release should be made shortly after the changes are pushed to master. The new release must include notes in `CHANGELOG.md` that describe how the API has been changed. The aim is to make it clear to users how their CCL-based code might be affected if they update to the latest version, especially if the update is likely to break something. All API changes should be discussed with the CCL team before being pushed to the `master` branch.

CHAPTER 7

Navigating the Code

The CCL package is laid out as follows.

- `.travis/`: helper scripts for running tests on Travis-CI
- `benchmarks/`: source code and data for the benchmark tests, see *Writing and Running Benchmarks* for more details.
- `cmake/`: `CMake` modules for building the CCL `C` layer
- `doc/`: latex source for the CCL note and the CCL paper
- `examples/`: (possible outdated) examples of how to use CCL
- `include/`: CCL `C` layer header files
- `pyccl/`: the CCL `Python` package
- `readthedocs/`: the CCL `Read the Docs` source
- `src/`: the CCL `C` layer source code
- `setup.py`: the `Python` install script
- `CMakeLists.txt`: the `CMake` installation configuration
- `.travis.yml`: the Travis-CI configuration
- `CHANGELOG.md`: the CCL log of changes to the code

7.1 Locations of Core Cosmological Quantities

Here we describe briefly the locations of the computations of core cosmological quantities. Usually each of these `C` files has a corresponding `Python` file, `SWIG` interface file, and `C` header file. However this is not always true for various reasons.

- `ccl_background.c`: computations of growth functions, the Hubble function and distances
- `ccl_core.c`: functions to handle cosmology structures and parameters

- ccl_error.c: functions to help handle C-layer errors
- ccl_bbks.c: the BBKS transfer function
- ccl_f1d.c: 1-d interpolations in C
- ccl_f2d.c: 2-d interpolations in C
- ccl_power.c: code to spline and initialize the power spectra and transfer functions
- ccl_bcm.c: code to compute the BCM model for baryonic effects
- ccl_eh.c: code to compute the Eisenstein and Hu (1998) transfer function approximation
- ccl_musigma.c: code to properly normalize input linear power spectra for the mu-Sigma modified gravity model
- ccl_cls.c: code to compute angular power spectra from 3-d power spectra
- ccl_massfunc.c: code for the halo mass function and halo bias models, also contains code to spline the linear power spectrum variance in top-hat windows sigma(R)
- *ccl_neutrinos.c*‘: code to compute the neutrino masses from their cosmological density and vice versa.
- ccl_emu17.c: code to compute the Cosmic Emu emulator for the matter power spectrum
- ccl_correlation.c: code to compute correlation functions from 2-d and 3-d power spectra
- ccl_halomod.c: code to compute halo model approximations to the power spctrum, also contains models for the halo mass-concentration relationship
- ccl_halofit.c: code to compute the HALOFIT approximation for the non-linear matter power spectrum
- ccl_haloprofile.c: code to compute common approximations to halo mass density profiles
- ccl_tracers.c: code to compute various kernels for tracers of large-scale structure (e.g., weak lensing kernels, galaxy clustering kernels, etc.)
- fftlog.c: an implementation of the FFTLog algorithm for fast transforms between Fourier and real space quantities

CHAPTER 8

Understanding the Python-C Interface

The Python interface to the CCL C code is built using SWIG, which automatically scans the CCL C headers and builds a matching interface in Python. The autogenerated SWIG interface can be accessed through the pyccl.ccllib module. The actual CCL API in Python is a more user-friendly wrapper that has been written on top of the SWIG-generated code.

8.1 Key Parts of SWIG Wrapper

The key parts/conventions of the SWIG wrapper are as follows:

- Interface .i files: These are kept in the pyccl/ directory, and tell SWIG which functions to extract from the C headers. There are also commands in these files to generate basic function argument documentation, and to remove the ccl_ prefix from function names. The interface files also contain code that tells SWIG how to convert C array arguments to numpy arrays. For certain functions, this code may also contain a simple loop to vectorize the function.
- Main interface file pyccl/ccl.i: This file imports all of the other interface files. Most of the CCL source files (e.g. ccl_core.c) have their own interface file too. For other files, mostly containing support/utility functions, SWIG only needs the C header (.h) file to be specified in the main ccl.i file.
- Internal CCL C-level global state: All global state in the CCL library (e.g., whether or not the power spectra splines are initialized and calling the function to initialize them) is controlled by the Python code. This convention ensures that when the C code is running, all data needed already exists. It also allows us to more easily use OpenMP in the C layer, avoiding the problem of having to manage the global state from more than one thread. In practice, this means that before you call the CCL C library from Python, you need to check to make sure that any internal C data you need has been initialized. The pyccl.Cosmology object has methods to help with this (e.g., compute_distances()).

8.2 Error Handling

Errors in CCL happen both in the `C` and `Python` code. For the `Python` code, one should simply raise the appropriate error, often `CCLError`. Error handling in the `C` code is a bit more subtle because the `C` code cannot (currently) raise `Python` exceptions directly.

Instead, CCL takes the convention that each `C` function passes back or returns a `status` value. Non-zero values indicate an error. Further, certain values have special meanings that are defined in the include/ccl_error.h header file. The `Python` file pyccl/_types.py contains translations of the `C` header macros into `Python` strings. These translations are used to translate the numerical error codes returned by the `C` code into something more human readable.

Further, in the CCL `C` layer, functions can write information about errors into a string held by the `C` version of the `pyccl.Cosmology` structure. The `Python` layer can then read the value of this string and raise it with a `Python` error when a non-zero status is returned.

Using these two mechanisms, `Python` functions in the CCL `Python` API can check for errors in the `C` layer and report back useful information to the user.

8.3 Example `Python` and `C` Code

Putting together all of the information above, here is example code based on the CCL `pyccl.bcm` module. Note that the code snippets here may be out of date relative to their current form in the repository.

8.3.1 C Header File `include/ccl_bcm.h`

```
/** @file */
#ifndef __CCL_BCM_H_INCLUDED__
#define __CCL_BCM_H_INCLUDED__

CCL_BEGIN_DECLS

/**
 * Correction for the impact of baryonic physics on the matter power spectrum.
 * Returns f(k,a) [dimensionless] for given cosmology, using the method specified for␣
↪the baryonic transfer function.
 * f(k,a) is the fractional change in the nonlinear matter power spectrum from the␣
↪Baryon Correction Model (BCM) of Schenider & Teyssier (2015). The parameters of the␣
↪model are passed as part of the cosmology class.
 * @param cosmo Cosmology parameters and configurations, including baryonic␣
↪parameters.
 * @param k Fourier mode, in [1/Mpc] units
 * @param a scale factor, normalized to 1 for today
 * @param status Status flag. 0 if there are no errors, nonzero otherwise.
 * For specific cases see documentation for ccl_error.c
 * @return f(k,a).
 */
double ccl_bcm_model_fka(ccl_cosmology * cosmo, double k, double a, int *status);

CCL_END_DECLS

#endif
```

Notice that the first argument is a C-level cosmology structure, the order of the arguments for wavenumber and scale factor, and that the last argument is a pointer to the `status` variable.

This C header file is then included in the `include/ccl.h` header file via `#include "ccl_bcm.h"`.

8.3.2 C Source File `src/ccl_bcm.c`

```
#include <math.h>
#include "ccl.h"

/* BCM correction */
// See Schneider & Teyssier (2015) for details of the model.
double ccl_bcm_model_fka(ccl_cosmology * cosmo, double k, double a, int *status) {
  double fkz;
  double b0;
  double bfunc, bfunc4;
  double kg;
  double gf,scomp;
  double kh;
  double z;

  if (a < 0) {
    *status = CCL_ERROR_PARAMETERS;
    return NaN;
  }

  // compute the BCM model here
  fkz = ...
  return fkz;
}
```

Here we see that if we encounter an error, the `status` variable is set and some fiducial value is returned. (Note that the check above does not actually exist in the main CCL source file.)

8.3.3 `SWIG` Interface File `pyccl/ccl_bcm.i`

```
%module ccl_bcm

%{
/* put additional #include here */
%}

// Enable vectorised arguments for arrays
%apply (double* IN_ARRAY1, int DIM1) {(double* k, int nk)};
%apply (int DIM1, double* ARGOUT_ARRAY1) {(int nout, double* output)};

%include "../include/ccl_bcm.h"

/* The python code here will be executed before all of the functions that
   follow this directive. */
%feature("pythonprepend") %{
    if numpy.shape(k) != (nout,):
        raise CCLError("Input shape for `k` must match `(nout,)`!")
%}

%inline %{
void bcm_model_fka_vec(ccl_cosmology * cosmo, double a, double* k, int nk,
                       int nout, double* output, int* status) {
```

(continues on next page)

(continued from previous page)

```
    for(int i=0; i < nk; i++){
      output[i] = ccl_bcm_model_fka(cosmo, k[i], a, status);
    }
}

%}

/* The directive gets carried between files, so we reset it at the end. */
%feature("pythonprepend") %{ %}
```

This SWIG interface file contains several examples of important features in writing SWIG interfaces.

- We have declared some inline C code to enable fast, vectorized computations of the BCM model.
- We have used the numpy.i type signatures, described in their documentation to enable vectorized inputs and outputs. The pyccl/ccl.i interface file defines a global numpy type signature for the status variable.
- We have used the SWIG pythonprepend feature to add a check on the sizes of the input and output arrays to ensure that we do not access memory out of bounds.
- We made sure the module name at the top matches the C source file.
- We made sure to include the C header file from the right relative path.

Finally, this SWIG interface file is included in the pyccl/ccl.i interface file.

8.3.4 Python Module pyccl/bcm.py

```
from . import ccllib as lib
from .pyutils import _vectorize_fn2


def bcm_model_fka(cosmo, k, a):
    """The BCM model correction factor for baryons.

    .. note:: BCM stands for the "baryonic correction model" of Schneider &
              Teyssier (2015; https://arxiv.org/abs/1510.06034). See the
              `DESC Note <https://github.com/LSSTDESC/CCL/blob/master/doc\
/0000-ccl_note/main.pdf>`_
              for details.

    .. note:: The correction factor is applied multiplicatively so that
              `P_corrected(k, a) = P(k, a) * factor(k, a)`.

    Args:
        cosmo (:obj:`Cosmology`): Cosmological parameters.
        k (float or array_like): Wavenumber; Mpc^-1.
        a (float): Scale factor.

    Returns:
        float or array_like: Correction factor to apply to the power spectrum.
    """
    return _vectorize_fn2(lib.bcm_model_fka,
                          lib.bcm_model_fka_vec, cosmo, k, a)
```

This file defines the actual API for the BCM model in CCL. It is the function signature and location of this function, along with what values it is supposed to return, that defines the API. Changes to any of the underlying C code or even the helper Python functions does not constitute an API breakage.

There are a few other features to note.

- We have written a complete, `Sphinx`-compatible docstring on the module.
- We are using the CCL helper function `_vectorize_fn2` to call the `SWIG`-generated interfaces `pyccl.ccllib.bcm_model_fka` and `pyccl.ccllib.bcm_model_fka_vec`.
- If any data needed to be initialized before calling the `SWIG` interfaces, we would need to initialize it by calling a function before `vectorize_fn2`. See the `pyccl.background` module for an example.

Finally, let's take a look at the implementation of the function `vectorize_fn2`

```
def _vectorize_fn2(fn, fn_vec, cosmo, x, z, returns_status=True):

    """Generic wrapper to allow vectorized (1D array) access to CCL functions with
    one vector argument and one scalar argument, with a cosmology dependence.

    Args:
        fn (callable): Function with a single argument.
        fn_vec (callable): Function that has a vectorized implementation in
                           a .i file.
        cosmo (ccl_cosmology or Cosmology): The input cosmology which gets
                                            converted to a ccl_cosmology.
        x (float or array_like): Argument to fn.
        z (float): Scalar argument to fn.
        returns_stats (bool): Indicates whether fn returns a status.

    """
    # Access ccl_cosmology object
    cosmo_in = cosmo
    cosmo = cosmo.cosmo
    status = 0

    # If a scalar was passed, convert to an array
    if isinstance(x, int):
        x = float(x)
    if isinstance(x, float):
        # Use single-value function
        if returns_status:
            f, status = fn(cosmo, x, z, status)
        else:
            f = fn(cosmo, x, z)
    elif isinstance(x, np.ndarray):
        # Use vectorised function
        if returns_status:
            f, status = fn_vec(cosmo, z, x, x.size, status)
        else:
            f = fn_vec(cosmo, z, x, x.size)
    else:
        # Use vectorised function
        if returns_status:
            f, status = fn_vec(cosmo, z, x, len(x), status)
        else:
            f = fn_vec(cosmo, z, x, len(x))

    # Check result and return
    check(status, cosmo_in)
    return f
```

This function does a few nice things for us

- It handles multiple input argument types, casting them to the right type to be passed to the SWIG-generated interface functions.
- At the end, it calls another function check. It is this function that checks the status variable and if it is non-zero, raises an error using the error string passed back from the C library.

You will find quite a few versions of this function in pyccl.pyutils for use in calling the SWIG-generated API for different numbers of arguments.

CHAPTER 9

Developer Installation

To develop new code for CCL, you need to install it using `pip`'s development install. This installation method creates a symbolic link between your `Python` `site-packages` directory and the copy of CCL you are working on locally. Thus when you change CCL `Python` code, you do not need to reinstall CCL in order for the changes to be visible system-wide. Note, if you change the CCL `C` code, you will need to force CCL to recompile the code (and copy the resulting `.so` into the `Python` package) by rerunning the command below.

To install CCL using a `pip` developer installation, you can execute

```
$ pip install -e .
```

from the top-level directory in the repository. You will need `CMake` in order to install CCL in this way. See *Getting CMake* for help installing `CMake` if you do not already have it. In order to run the tests, you will need both `CAMB` and `CLASS` installed. See the instructions for *Getting a Boltzmann Code* for details.

9.1 C-layer Dependencies and Requirements

CCL has several C dependencies. The `CMake` build will download and compile these automatically if they are not present on your system. However, if you do have them installed locally in a spot accessible to `CMake`, the local versions will be used.

These dependencies are

- GNU Scientific Library GSL, version 2.1 or above
- FFTW3 version 3.1 or above
- CLASS version 2.6.3 or above
- SWIG

9.2 Uninstalling CCL in Developer Mode

To uninstall CCL in developer mode, simply type

```
$ pip uninstall pyccl
```

CHAPTER 10

Development Workflow

CCL development proceeds via pull requests on GitHub. In short, the code base is forked on GitHub, a new change is pushed to a new branch, and then a pull request is issued by the developer. Once a pull request is issued, CCL uses Travis-CI to automatically run the benchmark/unit tests and the linter. Please read the sections below for guidance on how to modify the source code, test, and then document your modifications.

10.1 Working on the Python library

After completing the *Developer Installation*, you can change the `Python` library in your local copy of CCL. Any changes you make will be visible in `Python` after you restart your interpreter. A typical workflow is to make some change, rerun a failing benchmark or unit test, etc. until you've completed your new feature and/or fixed a bug. See *Writing and Running Benchmarks* and *Writing and Running Unit Tests* for instructions on how to run these test suites. Please add unit tests and benchmarks as appropriate for any new functionality. You should also make sure to run the linter by following the instructions in *Linting with flake8*. Finally, after you are satisfied with your new code, make sure to follow *Updating the Changelog and Documentation* in order to the properly document your new additions.

10.2 Adding New C-level Features to CCL

Before adding new features to the CCL `C` code, please read *Understanding the Python-C Interface* in order to understand how the `Python` and `C` codes interact.

The following steps are needed to add a new source file `ccl_newfile.c` and header `ccl_newfile.h` to the library.

1. Put `ccl_newfile.c` in the `src` directory.
2. Put `ccl_newfile.h` in the `include` directory.
3. Make sure to add a C-level include of your new header to `include/ccl.h`.
4. Add the new source file `ccl_newfile.c` to the `CCL_SRC` variable in `CMakeLists.txt`.

5. Write a `ccl_newfile.i SWIG` interface file for the new functions. This file gets placed in the `pyccl` directory. You may need to consult another CCL developer for help with the `SWIG` interface files.
6. Add your new `SWIG` interface file `ccl_newfile.i` to the list of includes in the `pyccl/ccl.i` file.
7. Write the public `Python` APIs for your new function in a file called `pyccl/newfile.py`. This `Python` package should access the `C` code you wrote from the new functions in `pyccl.ccllib`.
8. Import your new functions into the `pyccl/__init__.py` file as needed.
9. Add unit tests and benchmarks as appropriate for the new code. See *Writing and Running Benchmarks* and *Writing and Running Unit Tests* for details.
10. Make sure to run the linter by following the instructions in *Linting with flake8*.
11. Follow *Updating the Changelog and Documentation* in order to the properly document your new additions.

Occasionally, modifications made correctly as described above will still not function properly. This might be due to multiple `pyccl` installation files not being properly overwritten, and the `Python` interpreter getting confused. At this point it is often best to resort to the "nuclear option" of deleting all `Python` files related to `pyccl` and starting from scratch. The procedure is as follows:

1. Execute `pip uninstall pyccl` (possibly multiple times) to remove all installed versions of `pyccl`.
2. Delete the compiled `C` files.
   - Execute `rm -r build/` to delete the build directory.
   - Execute `rm pyccl/ccl_wrap.c pyccl/ccllib.py` to delete the autogenerated `SWIG` files.
   - Check various locations for the `Python site-packages` directory for any `pyccl` files and delete them.
3. Reinstall CCL according to the *Developer Installation* instructions.

10.3 Updating the Changelog and Documentation

Please follow the following guidelines for documenting changes to CCL:

- New features and bug fixes should be documented in the `CHANGELOG.md` with a brief description and the GitHub pull request number.
- Any new derivations/math essential to understanding a new CCL feature should be documented in the CCL note. See :ref:`cclnote` for instructions on how to modify and compile it.
- All additions to the `Python` public API should have `Python` docstrings. These are written in `Sphinx` compatible format so that they can be incorporated into the CCL `Read the Docs` pages. See the current `Python` docstrings in the `Python` source for examples.
- Additions to the `C` code should be documented/commented so that other CCL developers can understand the code.

10.4 Linting with `flake8`

CCL uses `flake8` to ensure that the `Python` code has a consistent style. `flake8` is available via `pip` or `conda` via `[pip|conda] install flake8`. You can run this tool locally by executing

```
$ flake8 pyccl
```

Any problems will be printed to `STDOUT`. No output indicates that `flake8` has succeeded.

10.5 Debug mode in Python

Because of the way the `Python` wrapper handles exceptions that occur inside the `C` code, by default users will only see error messages for the most recent error that occurs. If multiple errors occurred during a CCL function call, all but the most recent error message will be overwritten. This convention can make it difficult to debug the root cause of a problem.

To help with debugging this kind of issue, you can enable debug mode in the `Python` wrapper. To do so, simply call `pyccl.debug_mode(True)`. This will cause the `Python` wrapper to print all `C` error messages to `STDERR` whenever they occur. `Python` exceptions will only be raised for the most recent error, as before. (Note that Jupyter notebooks do not print `C STDERR` messages by default.)

10.6 Continuous Integration with Travis-CI

Travis-CI is a continuous integration service that reads the file `.travis.yml` file in the repository and then runs the benchmarks/unit tests. More details on Travis-CI can be found here: https://docs.travis-ci.com/user/getting-started/.

Every time you push a commit, Travis-CI will automatically try to build the libraries with your new changes and run the benchmark/unit tests. You can check the status of your builds by following the links from the pull request page. If your build errors or fails, you can scroll through the log to find out what went wrong. Warnings from `flake8` will result in the tests not passing. If your additions require new dependencies, make sure that you include them in the `conda` environments defined in `.travis/install.sh`.

10.7 Deploying a New Release

When cutting a new release, the procedure is as follows:

1. Make sure any API changes are documented in `CHANGELOG.md`
2. Commit to master
3. Create a new release from the GitHub interface here: https://github.com/LSSTDESC/CCL/releases/new
4. Manually create a source distribution from the root CCL folder:

   ```
   $ python setup.py sdist
   ```

   This command will create a `.tar.gz` file in the `dist` folder.
5. Upload source distribution to PyPi using `twine` (can be installed using `pip` or `conda`):

   ```
   $ twine upload  dist/pyccl-x.x.x.tar.gz
   ```

   Make sure your `twine` and `setuptools` packages are up to date, otherwise the markdown formatting of the `README.md` will not be correctly processed on the CCL PyPi page.
6. The `conda-forge` automated release bots will detect the new PyPi release and automatically make a pull request on the CCL feedstock. Once this pull request is merged, the new CCL release will be available on `conda` after a few hours.
7. Rebuild and redeploy the `Read the Docs` pages per the instructions in *Building the Read the Docs Documentation*. Note that you may need to adjust the major version number in `readthedocs/conf.py` if the new version has a major version number bump.

A CCL administrators will need to complete the steps above.

10.8 Building the Read the Docs Documentation

To build the Read the Docs documentation, follow the following steps:

```
$ cd readthedocs
$ make clean
$ make html
```

You can then inspect the outputs in readthedocs/_build/index.html to make sure the formatting is correct. Finally, contact the CCL administrators to redeploy the live documentation.

10.9 Building the CCL Note

The CCL note is a latex'ed documented located in doc/0000-ccl_note. It is used to document the scientific content of the CCL library. Note that documentation of the actual APIs and code should reside in the Python doc strings and other code comments.

To compile the CCL note, type make in the doc/0000-ccl_note directory.

If you need to modify the note, the files to modify are:

- authors.csv: To document your contribution.
- main.tex: To detail the changes to the library.
- main.bib: To add new references.

CHAPTER 11

Writing and Running Unit Tests

CCL uses the `pytest` package to run a suite of unit tests. This package can be installed via `pip` or `conda` with `[pip|conda] install pytest`.

11.1 Running the Unit Tests

To run the unit tests, execute

```
pytest -vv pyccl
```

from the top-level repository directory. Any errors will be reported at the end.

Other useful `pytest` options include

- `-k <name of test>`: adding this option with the name of the test will force `pytest` to only run that test
- `-s`: this option makes `pytest` print all output to STDOUT instead of its usual behavior of suppressing output
- `-x`: this option forces `pytest` to stop after the first failed test
- `--pdb`: this option will launch the `Python` debugger for failed tests, usually you want to combine it with `-s -x -k <name of test>` in order to debug a single failed test

11.2 Writing Unit Tests

Please follow the following guidelines when writing unit tests.

1. All unit tests should be written as modules in the `pytest/tests` submodule.
2. Each unit test is a function that does some operation with CCL and then uses a `Python` `assert` statement. A unit test is marked by `pytest` as failed when the `assert` statement fails.
3. The unit tests should be fast, ideally less than a second or so. This requirement means avoiding running `CLASS` or `CAMB` when it is not needed.

4. The unit tests file should have a name that matches the python module it is testing (e.g., `pyccl/tests/test_bcm.py` for `pyccl/bcm.py`).
5. The unit test function itself should have a name that starts with `test_` and is also descriptive.

An example of a unit test would be a file at `pyccl/tests/test_core.py` with the test

```
import numpy as np
import pyccl

def test_cosmology_sigma8_consistent():
    cosmo = pyccl.Cosmology(
        Omega_c=0.25, Omega_b=0.05, h=0.7, sigma8=0.8, n_s=0.95,
        transfer_function='bbks')
    assert np.allclose(pyccl.sigma8(cosmo), 0.8)
```

11.3 Building and Running CCL's C Unit Tests

CCL has a few vestigial unit tests written in `C`. Once you have `CMake` installed (see *Getting CMake*), you can build them with the following commands from the top-level CCL directory

```
$ mkdir -p build
$ cd build
$ cmake ..
$ make check_ccl
```

Then you can run the tests via

```
$ ./check_ccl
```

CHAPTER 12

Writing and Running Benchmarks

Nearly every new feature in CCL is benchmarked against an independent implementation. If you are adding a new feature, make sure someone can provide you with an independent output for cross-checks. Independent codes should be publicly available and a script to run them should be provided such that the user can reproduce the benchmarks.

12.1 Running the Benchmarks

After CCL is installed, the benchmarks can be run by executing

```
$ pytest -vv benchmarks
```

from the top-level directory of the repository. The *Writing and Running Unit Tests* section contains useful hints on using `pytest` to debug individual tests/benchmarks.

12.2 Writing a Benchmark

Benchmarks in CCL follow the `pytest` conventions for the CCL unit tests, but are located in the `benchmarks/` directory. Any data needed for the benchmark should be added to the `benchmarks/data` directory. Code to produce the benchmark data can be put in `benchmarks/data/code`. If it cannot be included in the repository, then a link on the wiki is acceptable.

The benchmark itself should read the data in the benchmarks data directory and then compare to the same computation done in CCL. Here is an example benchmark testing the CCL BBKS transfer function (located in the file `benchmarks/test_bbks.py`)

```
import numpy as np
import pyccl as ccl
import pytest

BBKS_TOLERANCE = 1.0e-5
```

(continues on next page)

(continued from previous page)

```
@pytest.mark.parametrize(
    'model,w0,wa',
    [(1, -1.0, 0.0),
     (2, -0.9, 0.0),
     (3, -0.9, 0.1)])
def test_bbks(model, w0, wa):
    cosmo = ccl.Cosmology(
        Omega_c=0.25,
        Omega_b=0.05,
        h=0.7,
        sigma8=0.8,
        n_s=0.96,
        Neff=0,
        m_nu=0.0,
        w0=w0,
        wa=wa,
        T_CMB=2.7,
        m_nu_type='normal',
        Omega_g=0,
        Omega_k=0,
        transfer_function='bbks',
        matter_power_spectrum='linear')

    data = np.loadtxt('./benchmarks/data/model%d_pk.txt' % model)

    k = data[:, 0] * cosmo['h']
    for i in range(6):
        a = 1.0 / (1.0 + i)
        pk = data[:, i+1] / (cosmo['h']**3)
        pk_ccl = ccl.linear_matter_power(cosmo, k, a)
        err = np.abs(pk_ccl/pk - 1)
        assert np.allclose(err, 0, rtol=0, atol=BBKS_TOLERANCE)
```

Note that the benchmarks are executed from the top-level of the repository so that the file paths for data are `benchmarks/data/<data file>`.

CHAPTER 13

Notation, Models and Other Cosmological Conventions

The documentation here is provides a brief description of CCL and its contents. For a more thorough description of the underlying equations CCL implements, see the CCL note and the CCL paper.

13.1 Cosmological Parameters

CCL uses the following parameters to define the cosmological model.

13.1.1 Background Parameters

- `Omega_c`: the density fraction at z=0 of CDM
- `Omega_b`: the density fraction at z=0 of baryons
- `h`: the Hubble constant in units of 100 Mpc/km/s
- `Omega_k`: the curvature density fraction at z=0
- `Omega_g`: the density of radiation (not including massless neutrinos)
- `w0`: first order term of the dark energy equation of state
- `wa`: second order term of the dark energy equation of state

13.1.2 Power Spectrum Normalization

The power spectrum normalization is given either as `A_s` (i.e., the primordial amplitude) or as `sigma8` (i.e., a measure of the amplitude today). Note that not all transfer functions support specifying a primordial amplitude.

- `sigma8`: the normalization of the power spectrum today, given by the RMS variance in spheres of 8 Mpc/h
- `A_s`: the primordial normalization of the power spectrum at k=0.05 Mpc $^{-1}$

13.1.3 Relativistic Species

- `Neff`: effective number of massless+massive neutrinos present at recombination
- `m_nu`: the total mass of massive neutrinos or the masses of the massive neutrinos in eV
- `m_nu_type`: how to interpret the `m_nu` argument, see the *options* below
- `T_CMB`: the temperature of the CMB today

13.1.4 Modified Gravity Parameters

- `mu_0` and `sigma_0`: the parameters of the scale-independent $\mu - \Sigma$ modified gravity model
- `df_mg` and `z_mg`: arrays of perturbations to the GR growth rate `f` as a function of redshift

13.2 Supported Models for the Power Spectrum, Mass Function, etc.

`pyccl` accepts strings indicating which model to use for various physical quantities (e.g., the transfer function). The various options are as follows.

`transfer_function` options

- `None` : do not compute a linear power spectrum
- 'eisenstein_hu': the Eisenstein and Hu (1998) fitting function
- 'bbks': the BBKS approximation
- 'boltzmann_class': use CLASS to compute the transfer function
- 'boltzmann_camb': use CAMB to compute the transfer function (default)

`matter_power_spectrum` options

- 'halo_model': use a halo model
- 'halofit': use HALOFIT (default)
- 'linear': neglect non-linear power spectrum contributions
- 'emu': use the Cosmic Emu

`baryons_power_spectrum` options

- 'nobaryons': neglect baryonic contributions to the power spectrum (default)
- 'bcm': use the baryonic correction model

`mass_function` options

- 'tinker': the Tinker et al. (2008) mass function
- 'tinker10': the Tinker et al. (2010) mass function (default)
- 'watson': the Watson et al. mass function
- 'angulo': the Angulo et al. mass function
- 'shethtormen': the Sheth and Tormen mass function

`halo_concentration` options

- 'bhattacharya2011': Bhattacharya et al. (2011) relation

- 'duffy2008': Duffy et al. (2008) relation (default)
- 'constant_concentration': use a constant concentration

`m_nu_type` options

This parameter specifies the model for massive neutrinos.

- 'list': specify each mass yourself in eV
- 'normal': use the normal hierarchy to convert total mass to individual masses (default)
- 'inverted': use the inverted hierarchy to convert total mass to individual masses
- 'equal': assume equal masses when converting the total mass to individual masses

`emulator_neutrinos` options

This parameter specifies how to handle inconsistencies in the treatment of neutrinos between the Cosmic Emu (equal masses) and other models.

- 'strict': fail unless things are absolutely consistent (default)
- 'equalize': redistribute the total mass equaly before using the Cosmic Emu. This option may result in slight internal inconsistencies in the physical model assumed for neutrinos.

13.3 Controlling Splines and Numerical Accuracy

The internal splines and integration accuracy are controlled by the attributes of `Cosmology.cosmo.spline_params` and `Cosmology.cosmo.gsl_params`. These should be set after instantiation, but before the object is used. For example, you can set the generic relative accuracy for integration by executing `c = Cosmology(...); c.cosmo.gsl_params.INTEGRATION_EPSREL = 1e-5`. The default values for these parameters are located in `src/ccl_core.c`.

The internal splines are controlled by the following parameters.

- A_SPLINE_NLOG: the number of logarithmically spaced bins between A_SPLINE_MINLOG and A_SPLINE_MIN.
- A_SPLINE_NA: the number of linearly spaced bins between A_SPLINE_MIN and A_SPLINE_MAX.
- A_SPLINE_MINLOG: the minimum value of the scale factor splines used for distances, etc.
- A_SPLINE_MIN: the transition scale factor between logarithmically spaced spline points and linearly spaced spline points.
- A_SPLINE_MAX: the the maximum value of the scale factor splines used for distances, etc.
- LOGM_SPLINE_NM: the number of logarithmically spaced values in mass for splines used in the computation of the halo mass function.
- LOGM_SPLINE_MIN: the base-10 logarithm of the minimum halo mass for splines used in the computation of the halo mass function.
- LOGM_SPLINE_MAX: the base-10 logarithm of the maximum halo mass for splines used in the computation of the halo mass function.
- LOGM_SPLINE_DELTA: the step in base-10 logarithmic units for computing finite difference derivatives in the computation of the mass function.
- A_SPLINE_NLOG_PK: the number of logarithmically spaced bins between A_SPLINE_MINLOG_PK and A_SPLINE_MIN_PK.
- A_SPLINE_NA_PK: the number of linearly spaced bins between A_SPLINE_MIN_PK and A_SPLINE_MAX.

- A_SPLINE_MINLOG_PK: the minimum value of the scale factor used for the power spectrum splines.
- A_SPLINE_MIN_PK: the transition scale factor between logarithmically spaced spline points and linearly spaced spline points for the power spectrum.
- K_MIN: the minimum wavenumber for the power spectrum splines for analytic models (e.g., BBKS, Eisenstein & Hu, etc.).
- K_MAX: the maximum wavenumber for the power spectrum splines for analytic models (e.g., BBKS, Eisenstein & Hu, etc.).
- K_MAX_SPLINE: the maximum wavenumber for the power spectrum splines for numerical models (e.g., ComsicEmu, CLASS, etc.).
- N_K: the number of spline nodes per decade for the power spectrum splines.
- N_K_3DCOR: the number of spline points in wavenumber per decade used for computing the 3D correlation function.
- ELL_MIN_CORR: the minimum value of the spline in angular wavenumber for correlation function computations with FFTlog.
- ELL_MAX_CORR: the maximum value of the spline in angular wavenumber for correlation function computations with FFTlog.
- N_ELL_CORR: the number of logarithmically spaced bins in angular wavenumber between ELL_MIN_CORR and ELL_MAX_CORR.

The numrical accuracy of GSL computations are controlled by the following parameters.

- N_ITERATION: the size of the GSL workspace for numerical integration.
- INTEGRATION_GAUSS_KRONROD_POINTS: the Gauss-Kronrod quadrature rule used for adaptive integrations.
- INTEGRATION_EPSREL: the relative error tolerance for numerical integration; used if not specified by a more specific parameter.
- INTEGRATION_LIMBER_GAUSS_KRONROD_POINTS: the Gauss-Kronrod quadrature rule used for adaptive integrations on subintervals for Limber integrals.
- INTEGRATION_LIMBER_EPSREL: the relative error tolerance for numerical integration of Limber integrals.
- INTEGRATION_DISTANCE_EPSREL: the relative error tolerance for numerical integration of distance integrals.
- INTEGRATION_SIGMAR_EPSREL: the relative error tolerance for numerical integration of power spectrum variance intrgals for the mass function.
- ROOT_EPSREL: the relative error tolerance for root finding used to invert the relationship between comoving distance and scale factor.
- ROOT_N_ITERATION: the maximum number of iterations used to for root finding to invert the relationship between comoving distance and scale factor.
- ODE_GROWTH_EPSREL: the relative error tolerance for integrating the linear growth ODEs.
- EPS_SCALEFAC_GROWTH: 10x the starting step size for integrating the linear growth ODEs and the scale factor of the initial condition for the linear growth ODEs.
- HM_MMIN: the minimum mass for halo model integrations.
- HM_MMAX: the maximum mass for halo model integrations.
- HM_EPSABS: the absolute error tolerance for halo model integrations.

- HM_EPSREL: the relative error tolerance for halo model integrations.
- HM_LIMIT: the size of the GSL workspace for halo moodel integrations.
- HM_INT_METHOD: the Gauss-Kronrod quadrature rule used for adaptive integrations for the halo model comptutations.

13.4 Specifying Physical Constants

The values of physical constants are set globally. These can be changed by assigning a new value to the attributes of `pyccl.physical_constants`. The following constants are defined and their default values are located in `src/ccl_core.c`. Note that the neutrino mass splittings are taken from Lesgourgues & Pastor (2012; 1212.6154). Also, see the CCL note for a discussion of the values of these constants from different sources.

basic physical constants

- CLIGHT_HMPC: speed of light / H0 in units of Mpc/h
- GNEWT: Newton's gravitational constant in units of m^3/Kg/s^2
- SOLAR_MASS: solar mass in units of kg
- MPC_TO_METER: conversion factor for Mpc to meters.
- PC_TO_METER: conversion factor for parsecs to meters.
- RHO_CRITICAL: critical density in units of M_sun/h / (Mpc/h)^3
- KBOLTZ: Boltzmann constant in units of J/K
- STBOLTZ: Stefan-Boltzmann constant in units of kg/s^3 / K^4
- HPLANCK: Planck's constant in units kg m^2 / s
- CLIGHT: speed of light in m/s
- EV_IN_J: conversion factor between electron volts and Joules
- T_CMB: temperature of the CMB in K
- TNCDM: temperature of the cosmological neutrino background in K

neutrino mass splittings

- DELTAM12_sq: squared mass difference between eigenstates 2 and 1.
- DELTAM13_sq_pos: squared mass difference between eigenstates 3 and 1 for the normal hierarchy.
- DELTAM13_sq_neg: squared mass difference between eigenstates 3 and 1 for the inverted hierarchy.

CHAPTER 14

pyccl

14.1 pyccl package

14.1.1 Submodules

pyccl.background module

Smooth background quantities

CCL defines seven species types:

- 'matter': cold dark matter and baryons
- 'dark_energy': cosmological constant or otherwise
- 'radiation': relativistic species besides massless neutrinos (i.e., only photons)
- 'curvature': curvature density
- 'neutrinos_rel': relativistic neutrinos
- 'neutrinos_massive': massive neutrinos

These strings define the *species* inputs to the functions below.

pyccl.background.**Sig_MG**(*cosmo*, *a*)

Redshift-dependent modification to Poisson equation for massless particles under modified gravity.

Parameters

- **cosmo** (ccl.cosmology) – Cosmological parameters.
- **a** (*float or array_like*) – Scale factor(s), normalized to 1 today.

Returns

Modification to Poisson equation under modified gravity at scale factor a. Sig_MG is assumed to be proportional to Omega_Lambda(z), see e.g. Abbott et al. 2018, 1810.02499, Eq. 9.

Return type Sig_MG (float or array_like)

pyccl.background.**comoving_angular_distance**(*cosmo*, *a*)
Comoving angular distance.

Note: This quantity is otherwise known as the transverse comoving distance, and is NOT angular diameter distance or angular separation. The comovoing angular distance is defined such that the comoving distance between two objects at a fixed scale factor separated by an angle :math:‘ heta‘ is :math:‘ heta D_{T}(a)‘ where $D_T(a)$ is the comoving angular distance.

Parameters

- **cosmo** (Cosmology) – Cosmological parameters.
- **a** (*float or array_like*) – Scale factor(s), normalized to 1 today.

Returns Comoving angular distance; Mpc.

Return type float or array_like

pyccl.background.**comoving_radial_distance**(*cosmo*, *a*)
Comoving radial distance.

Parameters

- **cosmo** (Cosmology) – Cosmological parameters.
- **a** (*float or array_like*) – Scale factor(s), normalized to 1 today.

Returns Comoving radial distance; Mpc.

Return type float or array_like

pyccl.background.**distance_modulus**(*cosmo*, *a*)
Distance Modulus, defined as 5 * log(luminosity distance / 10 pc).

Note: The distance modulus can be used to convert between apparent and absolute magnitudes via m = M + distance modulus, where m is the apparent magnitude and M is the absolute magnitude.

Parameters

- **cosmo** (Cosmology) – Cosmological parameters.
- **a** (*float or array_like*) – Scale factor(s), normalized to 1 today.

Returns Distance modulus at a.

Return type float or array_like

pyccl.background.**growth_factor**(*cosmo*, *a*)
Growth factor.

Parameters

- **cosmo** (Cosmology) – Cosmological parameters.
- **a** (*float or array_like*) – Scale factor(s), normalized to 1 today.

Returns Growth factor, normalized to unity today.

Return type float or array_like

pyccl.background.**growth_factor_unnorm**(*cosmo*, *a*)
Unnormalized growth factor.

Parameters

- **cosmo** (Cosmology) – Cosmological parameters.
- **a** (*float or array_like*) – Scale factor(s), normalized to 1 today.

Returns

Unnormalized growth factor, normalized to the scale factor at early times.

Return type float or array_like

pyccl.background.**growth_rate**(*cosmo*, *a*)
Growth rate defined as the logarithmic derivative of the growth factor, dlnD/dlna.

Parameters

- **cosmo** (Cosmology) – Cosmological parameters.
- **a** (*float or array_like*) – Scale factor(s), normalized to 1 today.

Returns Growth rate.

Return type float or array_like

pyccl.background.**h_over_h0**(*cosmo*, *a*)
Ratio of Hubble constant at *a* over Hubble constant today.

Parameters

- **cosmo** (Cosmology) – Cosmological parameters.
- **a** (*float or array_like*) – Scale factor(s), normalized to 1 today.

Returns H(a)/H0.

Return type float or array_like

pyccl.background.**luminosity_distance**(*cosmo*, *a*)
Luminosity distance.

Parameters

- **cosmo** (Cosmology) – Cosmological parameters.
- **a** (*float or array_like*) – Scale factor(s), normalized to 1 today.

Returns Luminosity distance; Mpc.

Return type float or array_like

pyccl.background.**mu_MG**(*cosmo*, *a*)
Redshift-dependent modification to Poisson equation under modified gravity.

Parameters

- **cosmo** (ccl.cosmology) – Cosmological parameters.
- **a** (*float or array_like*) – Scale factor(s), normalized to 1 today.

Returns

Modification to Poisson equation under modified gravity at a scale factor. mu_MG is assumed to be proportional to Omega_Lambda(z), see e.g. Abbott et al. 2018, 1810.02499, Eq. 9.

Return type mu_MG (float or array_like)

pyccl.background.**omega_x**(*cosmo*, *a*, *species*)
Density fraction of a given species at a redshift different than z=0.

Parameters

- **cosmo** (Cosmology) – Cosmological parameters.
- **a** (*float or array_like*) – Scale factor(s), normalized to 1 today.
- **species** (*string*) – species type. Should be one of
  - 'matter': cold dark matter, massive neutrinos, and baryons
  - 'dark_energy': cosmological constant or otherwise
  - 'radiation': relativistic species besides massless neutrinos
  - 'curvature': curvature density
  - 'neutrinos_rel': relativistic neutrinos
  - 'neutrinos_massive': massive neutrinos

Returns

Density fraction of a given species at a scale factor.

Return type float or array_like

pyccl.background.**rho_x**(*cosmo*, *a*, *species*, *is_comoving=False*)
Physical or comoving density as a function of scale factor.

Parameters

- **cosmo** (Cosmology) – Cosmological parameters.
- **a** (*float or array_like*) – Scale factor(s), normalized to 1 today.
- **species** (*string*) – species type. Should be one of
  - 'matter': cold dark matter, massive neutrinos, and baryons
  - 'dark_energy': cosmological constant or otherwise
  - 'radiation': relativistic species besides massless neutrinos
  - 'curvature': curvature density
  - 'neutrinos_rel': relativistic neutrinos
  - 'neutrinos_massive': massive neutrinos
- **is_comoving** (*bool*) – either physical (False, default) or comoving (True)

Returns Physical density of a given species at a scale factor.

Return type rho_x (float or array_like)

pyccl.background.**scale_factor_of_chi**(*cosmo*, *chi*)
Scale factor, a, at a comoving radial distance chi.

Parameters

- **cosmo** (Cosmology) – Cosmological parameters.
- **chi** (*float or array_like*) – Comoving radial distance(s); Mpc.

Returns Scale factor(s), normalized to 1 today.

Return type float or array_like

pyccl.bcm module

pyccl.bcm.**bcm_model_fka**(*cosmo*, *k*, *a*)
The BCM model correction factor for baryons.

Note: BCM stands for the "baryonic correction model" of Schneider & Teyssier (2015; https://arxiv.org/abs/1510.06034). See the DESC Note for details.

Note: The correction factor is applied multiplicatively so that *P_corrected(k, a) = P(k, a) * factor(k, a)*.

Parameters

- **cosmo** (Cosmology) – Cosmological parameters.
- **k** (*float or array_like*) – Wavenumber; Mpc^-1.
- **a** (*float*) – Scale factor.

Returns Correction factor to apply to the power spectrum.

Return type float or array_like

pyccl.boltzmann module

pyccl.boltzmann.**get_camb_pk_lin**(*cosmo*)
Run CAMB and return the linear power spectrum.

Parameters **cosmo** (Cosmology) – Cosmological parameters. The cosmological parameters with which to run CAMB.

Returns

power spectrum object The linear power spectrum.

Return type pk_lin (Pk2D)

pyccl.boltzmann.**get_class_pk_lin**(*cosmo*)
Run CLASS and return the linear power spectrum.

Parameters **cosmo** (Cosmology) – Cosmological parameters. The cosmological parameters with which to run CLASS.

Returns

power spectrum object The linear power spectrum.

Return type pk_lin (Pk2D)

pyccl.cls module

pyccl.cls.**angular_cl**(*cosmo*, *cltracer1*, *cltracer2*, *ell*, *p_of_k_a=None*, *l_limber=-1.0*)
Calculate the angular (cross-)power spectrum for a pair of tracers.

Parameters

- **cosmo** (Cosmology) – A Cosmology object.
- **cltracer2** (*cltracer1,*) – Tracer objects, of any kind.

- **ell** (*float or array_like*) – Angular wavenumber(s) at which to evaluate the angular power spectrum.
- **p_of_k_a** (Pk2D or None) – 3D Power spectrum to project. If None, the non-linear matter power spectrum will be used.
- **l_limber** (*float*) – Angular wavenumber beyond which Limber's approximation will be used. Defaults to -1.

Returns

Angular (cross-)power spectrum values, C_ℓ, for the pair of tracers, as a function of ℓ.

Return type float or array_like

pyccl.core module

The core functionality of ccl, including the core data types. This includes the cosmology and parameters objects used to instantiate a model from which one can compute a set of theoretical predictions.

class pyccl.core.**Cosmology**(*Omega_c=None*, *Omega_b=None*, *h=None*, *n_s=None*, *sigma8=None*, *A_s=None*, *Omega_k=0.0*, *Omega_g=None*, *Neff=3.046*, *m_nu=0.0*, *m_nu_type=None*, *w0=-1.0*, *wa=0.0*, *T_CMB=None*, *bcm_log10Mc=14.079181246047625*, *bcm_etab=0.5*, *bcm_ks=55.0*, *mu_0=0.0*, *sigma_0=0.0*, *z_mg=None*, *df_mg=None*, *transfer_function='boltzmann_camb'*, *matter_power_spectrum='halofit'*, *baryons_power_spectrum='nobaryons'*, *mass_function='tinker10'*, *halo_concentration='duffy2008'*, *emulator_neutrinos='strict'*)

Bases: object

A cosmology including parameters and associated data.

Note: Although some arguments default to *None*, they will raise a ValueError inside this function if not specified, so they are not optional.

Note: The parameter Omega_g can be used to set the radiation density (not including relativistic neutrinos) to zero. Doing this will give you a model that is physically inconsistent since the temperature of the CMB will still be non-zero. Note however that this approximation is common for late-time LSS computations.

Note: BCM stands for the "baryonic correction model" of Schneider & Teyssier (2015; https://arxiv.org/abs/1510.06034). See the DESC Note for details.

Note: After instantiation, you can set parameters related to the internal splines and numerical integration accuracy by setting the values of the attributes of `Cosmology.cosmo.spline_params` and `Cosmology.cosmo.gsl_params`. For example, you can set the generic relative accuracy for integration by executing `c = Cosmology(...); c.cosmo.gsl_params.INTEGRATION_EPSREL = 1e-5`. See the module level documetaion of *pyccl.core* for details.

Parameters

- **Omega_c** (float) – Cold dark matter density fraction.
- **Omega_b** (float) – Baryonic matter density fraction.
- **h** (float) – Hubble constant divided by 100 km/s/Mpc; unitless.
- **A_s** (float) – Power spectrum normalization. Exactly one of A_s and sigma_8 is required.
- **sigma8** (float) – Variance of matter density perturbations at an 8 Mpc/h scale. Exactly one of A_s and sigma_8 is required.
- **n_s** (float) – Primordial scalar perturbation spectral index.
- **Omega_k** (float, optional) – Curvature density fraction. Defaults to 0.
- **Omega_g** (float, optional) – Density in relativistic species except massless neutrinos. The default of *None* corresponds to setting this from the CMB temperature. Note that if a non-*None* value is given, this may result in a physically inconsistent model because the CMB temperature will still be non-zero in the parameters.
- **Neff** (float, optional) – Effective number of massless neutrinos present. Defaults to 3.046.
- **m_nu** (float, optional) – Total mass in eV of the massive neutrinos present. Defaults to 0.
- **m_nu_type** (str, optional) – The type of massive neutrinos. Should be one of 'inverted', 'normal', 'equal' or 'list'. The default of None is the same as 'normal'.
- **w0** (float, optional) – First order term of dark energy equation of state. Defaults to -1.
- **wa** (float, optional) – Second order term of dark energy equation of state. Defaults to 0.
- **T_CMB** (float) – The CMB temperature today. The default of None uses the global CCL value in pyccl.physical_constants.T_CMB.
- **bcm_log10Mc** (float, optional) – One of the parameters of the BCM model. Defaults to *np.log10(1.2e14)*.
- **bcm_etab** (float, optional) – One of the parameters of the BCM model. Defaults to 0.5.
- **bcm_ks** (float, optional) – One of the parameters of the BCM model. Defaults to 55.0.
- **mu_0** (float, optional) – One of the parameters of the mu-Sigma modified gravity model. Defaults to 0.0
- **sigma_0** (float, optional) – One of the parameters of the mu-Sigma modified gravity model. Defaults to 0.0
- **df_mg** (*array_like, optional*) – Perturbations to the GR growth rate as a function of redshift Δf. Used to implement simple modified growth scenarios.
- **z_mg** (*array_like, optional*) – Array of redshifts corresponding to df_mg.
- **transfer_function** (str, optional) – The transfer function to use. Defaults to 'boltzmann_camb'.
- **matter_power_spectrum** (str, optional) – The matter power spectrum to use. Defaults to 'halofit'.
- **baryons_power_spectrum** (str, optional) – The correction from baryonic effects to be implemented. Defaults to 'nobaryons'.
- **mass_function** (str, optional) – The mass function to use. Defaults to 'tinker10' (2010).

- **halo_concentration** (str, optional) – The halo concentration relation to use. Defaults to Duffy et al. (2008) 'duffy2008'.
- **emulator_neutrinos** – *str*, optional): If using the emulator for the power spectrum, specified treatment of unequal neutrinos. Options are 'strict', which will raise an error and quit if the user fails to pass either a set of three equal masses or a sum with m_nu_type = 'equal', and 'equalize', which will redistribute masses to be equal right before calling the emualtor but results in internal inconsistencies. Defaults to 'strict'.

compute_distances()
Compute the distance splines.

compute_growth()
Compute the growth function.

compute_linear_power()
Compute the linear power spectrum.

compute_nonlin_power()
Compute the non-linear power spectrum.

compute_sigma()
Compute the sigma(M) and mass function splines.

has_distances
Checks if the distances have been precomputed.

has_growth
Checks if the growth function has been precomputed.

has_linear_power
Checks if the linear power spectra have been precomputed.

has_nonlin_power
Checks if the non-linear power spectra have been precomputed.

has_sigma
Checks if sigma(M) and mass function splines are precomputed.

classmethod read_yaml(*filename*)
Read the parameters from a YAML file.

Parameters **filename** (str) –

status()
Get error status of the ccl_cosmology object.

Note: The error statuses are currently under development and may not be fully descriptive.

Returns str containing the status message.

write_yaml(*filename*)
Write a YAML representation of the parameters to file.

Parameters **filename** (str) –

pyccl.correlation module

Correlation functon computations.

Choices of algorithms used to compute correlation functions: 'Bessel' is a direct integration using Bessel functions. 'FFTLog' is fast using a fast Fourier transform. 'Legendre' uses a sum over Legendre polynomials.

pyccl.correlation.**correlation**(*cosmo*, *ell*, *C_ell*, *theta*, *corr_type='gg'*, *method='fftlog'*)
Compute the angular correlation function.

Parameters

- **cosmo** (Cosmology) – A Cosmology object.
- **ell** (*array_like*) – Multipoles corresponding to the input angular power spectrum.
- **C_ell** (*array_like*) – Input angular power spectrum.
- **theta** (*float or array_like*) – Angular separation(s) at which to calculate the angular correlation function (in degrees).
- **corr_type** (*string*) – Type of correlation function. Choices: 'GG' (galaxy-galaxy), 'GL' (galaxy-shear), 'L+' (shear-shear, xi+), 'L-' (shear-shear, xi-).
- **method** (*string, optional*) – Method to compute the correlation function. Choices: 'Bessel' (direct integration over Bessel function), 'FFTLog' (fast integration with FFTLog), 'Legendre' (brute-force sum over Legendre polynomials).

Returns

Value(s) of the correlation function at the input angular separations.

Return type float or array_like

pyccl.correlation.**correlation_3d**(*cosmo*, *a*, *r*)
Compute the 3D correlation function.

Parameters

- **cosmo** (Cosmology) – A Cosmology object.
- **a** (*float*) – scale factor.
- **r** (*float or array_like*) – distance(s) at which to calculate the 3D correlation function (in Mpc).

Returns Value(s) of the correlation function at the input distance(s).

pyccl.correlation.**correlation_3dRsd**(*cosmo*, *a*, *s*, *mu*, *beta*, *use_spline=True*)
Compute the 3DRsd correlation function using linear approximation with multipoles.

Parameters

- **cosmo** (Cosmology) – A Cosmology object.
- **a** (*float*) – scale factor.
- **s** (*float or array_like*) – distance(s) at which to calculate the 3DRsd correlation function (in Mpc).
- **mu** (*float*) – cosine of the angle at which to calculate the 3DRsd correlation function (in Radian).
- **beta** (*float*) – growth rate divided by galaxy bias.
- **use_spline** – switch that determines whether the RSD correlation function is calculated using global splines of multipoles.

Returns Value(s) of the correlation function at the input distance(s) & angle.

pyccl.correlation.**correlation_3dRsd_avgmu**(*cosmo*, *a*, *s*, *beta*)
Compute the 3DRsd correlation function averaged over mu at constant s.

Parameters

- **cosmo** (Cosmology) – A Cosmology object.
- **a** (*float*) – scale factor.
- **s** (*float or array_like*) – distance(s) at which to calculate the 3DRsd correlation function (in Mpc).
- **beta** (*float*) – growth rate divided by galaxy bias.

Returns Value(s) of the correlation function at the input distance(s) & angle.

pyccl.correlation.**correlation_multipole**(*cosmo*, *a*, *beta*, *l*, *s*)
Compute the correlation multipoles.

Parameters

- **cosmo** (Cosmology) – A Cosmology object.
- **a** (*float*) – scale factor.
- **beta** (*float*) – growth rate divided by galaxy bias.
- **l** (*int*) – the desired multipole
- **s** (*float or array_like*) – distance(s) at which to calculate the 3DRsd correlation function (in Mpc).

Returns Value(s) of the correlation function at the input distance(s).

pyccl.correlation.**correlation_pi_sigma**(*cosmo*, *a*, *beta*, *pi*, *sig*, *use_spline=True*)
Compute the 3DRsd correlation in pi-sigma space.

Parameters

- **cosmo** (Cosmology) – A Cosmology object.
- **a** (*float*) – scale factor.
- **pi** (*float*) – distance times cosine of the angle (in Mpc).
- **sig** (*float or array-like*) – distance(s) times sine of the angle (in Mpc).
- **beta** (*float*) – growth rate divided by galaxy bias.

Returns Value(s) of the correlation function at the input pi and sigma.

pyccl.errors module

exception pyccl.errors.**CCLError**
Bases: RuntimeError

A CCL-specific RuntimeError

exception pyccl.errors.**CCLWarning**
Bases: RuntimeWarning

A CCL-specific warning

pyccl.halomodel module

pyccl.halomodel.**halo_concentration**(*cosmo*, *halo_mass*, *a*, *odelta=200*)
Halo mass concentration relation

Parameters

- **cosmo** (Cosmology) – Cosmological parameters.
- **halo_mass** (*float or array_like*) – Halo masses; Msun.
- **a** (*float*) – scale factor.
- **odelta** (*float*) – overdensity parameter (default: 200)

Returns Dimensionless halo concentration, ratio rv/rs

Return type float or array_like

pyccl.halomodel.**halomodel_matter_power**(*cosmo*, *k*, *a*)
Matter power spectrum from halo model assuming NFW density profiles :param cosmo: Cosmological parameters. :type cosmo: Cosmology :param a: scale factor. :type a: float :param k: wavenumber :type k: float or array_like

Returns

matter power spectrum from halo model

Return type halomodel_matter_power (float or array_like)

pyccl.halomodel.**onehalo_matter_power**(*cosmo*, *k*, *a*)
One-halo term for matter power spectrum assuming NFW density profiles :param cosmo: Cosmological parameters. :type cosmo: Cosmology :param a: scale factor. :type a: float :param k: wavenumber :type k: float or array_like

Returns

one-halo term for matter power spectrum

Return type onehalo_matter_power (float or array_like)

pyccl.halomodel.**twohalo_matter_power**(*cosmo*, *k*, *a*)
Two-halo term for matter power spectrum assuming NFW density profiles :param cosmo: Cosmological parameters. :type cosmo: Cosmology :param a: scale factor. :type a: float :param k: wavenumber :type k: float or array_like

Returns

two-halo term for matter power spectrum

Return type two-halo matter power spectrum (float or array_Like)

pyccl.haloprofile module

pyccl.haloprofile.**einasto_profile_3d**(*cosmo*, *concentration*, *halo_mass*, *odelta*, *a*, *r*)
Calculate the 3D Einasto halo profile at a given radius or an array of radii, for a halo with a given mass, mass definition, and concentration, at a given scale factor, with a cosmology dependence. The alpha parameter is calibrated using the relation with peak height in https://arxiv.org/pdf/1401.1216.pdf eqn5, assuming virial mass.

Parameters

- **cosmo** (Cosmology) – cosmological parameters.
- **concentration** (*float*) – halo concentration.

- **halo_mass** (*float*) – halo masses; in units of Msun.
- **odelta** (*float*) – overdensity with respect to mean matter density.
- **a** (*float*) – scale factor.
- **r** (*float or array_like*) – radius or radii to calculate profile for, in units of Mpc.

Returns 3D NFW density at r, in units of Msun/Mpc^3.

Return type float or array_like

pyccl.haloprofile.**hernquist_profile_3d**(*cosmo*, *concentration*, *halo_mass*, *odelta*, *a*, *r*)

Calculate the 3D Hernquist halo profile at a given radius or an array of radii, for a halo with a given mass, mass definition, and concentration, at a given scale factor, with a cosmology dependence.

Parameters

- **cosmo** (Cosmology) – cosmological parameters.
- **concentration** (*float*) – halo concentration.
- **halo_mass** (*float*) – halo masses; in units of Msun.
- **odelta** (*float*) – overdensity with respect to mean matter density.
- **a** (*float*) – scale factor.
- **r** (*float or array_like*) – radius or radii to calculate profile for, in units of Mpc.

Returns 3D NFW density at r, in units of Msun/Mpc^3.

Return type float or array_like

pyccl.haloprofile.**nfw_profile_2d**(*cosmo*, *concentration*, *halo_mass*, *odelta*, *a*, *r*)

Calculate the 2D projected NFW halo profile at a given radius or an array of radii, for a halo with a given mass, mass definition, and concentration, at a given scale factor, with a cosmology dependence.

Parameters

- **cosmo** (Cosmology) – cosmological parameters.
- **concentration** (*float*) – halo concentration.
- **halo_mass** (*float*) – halo masses; in units of Msun.
- **odelta** (*float*) – overdensity with respect to mean matter density.
- **a** (*float*) – scale factor.
- **r** (*float or array_like*) – radius or radii to calculate profile for, in units of Mpc.

Returns

2D projected NFW density at r, in units of Msun/Mpc^2.

Return type float or array_like

pyccl.haloprofile.**nfw_profile_3d**(*cosmo*, *concentration*, *halo_mass*, *odelta*, *a*, *r*)

Calculate the 3D NFW halo profile at a given radius or an array of radii, for a halo with a given mass, mass definition, and concentration, at a given scale factor, with a cosmology dependence.

Parameters

- **cosmo** (Cosmology) – cosmological parameters.
- **concentration** (*float*) – halo concentration.
- **halo_mass** (*float*) – halo masses; in units of Msun.

- **odelta** (*float*) – overdensity with respect to mean matter density.
- **a** (*float*) – scale factor.
- **r** (*float or array_like*) – radius or radii to calculate profile for, in units of Mpc.

Returns 3D NFW density at r, in units of Msun/Mpc^3.

Return type float or array_like

pyccl.massfunction module

pyccl.massfunction.**halo_bias**(*cosmo*, *halo_mass*, *a*, *overdensity=200*)
Tinker et al. (2010) halo bias

Parameters

- **cosmo** (Cosmology) – Cosmological parameters.
- **halo_mass** (*float or array_like*) – Halo masses; Msun.
- **a** (*float*) – Scale factor.
- **overdensity** (*float*) – Overdensity parameter (default: 200).

Returns Halo bias.

Return type float or array_like

pyccl.massfunction.**massfunc**(*cosmo*, *halo_mass*, *a*, *overdensity=200*)
Tinker et al. (2010) halo mass function, dn/dlog10M.

Parameters

- **cosmo** (Cosmology) – Cosmological parameters.
- **halo_mass** (*float or array_like*) – Halo masses; Msun.
- **a** (*float*) – scale factor.
- **overdensity** (*float*) – overdensity parameter (default: 200)

Returns Halo mass function; dn/dlog10M.

Return type float or array_like

pyccl.massfunction.**massfunc_m2r**(*cosmo*, *halo_mass*)
Converts smoothing halo mass into smoothing halo radius.

Note: This is R=(3M/(4*pi*rho_m))^(1/3), where rho_m is the mean matter density.

Parameters

- **cosmo** (Cosmology) – Cosmological parameters.
- **halo_mass** (*float or array_like*) – Halo masses; Msun.

Returns Smoothing halo radius; Mpc.

Return type float or array_like

pyccl.massfunction.**sigmaM**(*cosmo*, *halo_mass*, *a*)
Root mean squared variance for the given halo mass of the linear power spectrum; Msun.

Parameters

- **cosmo** (Cosmology) – Cosmological parameters.
- **halo_mass** (*float or array_like*) – Halo masses; Msun.
- **a** (*float*) – scale factor.

Returns RMS variance of halo mass.

Return type float or array_like

pyccl.neutrinos module

pyccl.neutrinos.**Omeganuh2**(*a*, *m_nu*, *T_CMB=None*)

Calculate Omega_nu * h^2 at a given scale factor given the sum of the neutrino masses.

Note: for all practical purposes, Neff is simply N_nu_mass.

Parameters

- **a** (*float or array-like*) – Scale factor, normalized to 1 today.
- **m_nu** (*float or array-like*) – Neutrino mass (in eV)
- **T_CMB** (*float, optional*) – Temperature of the CMB (K). Default: 2.725.

Returns corresponding to a given neutrino mass.

Return type float or array_like

pyccl.neutrinos.**nu_masses**(*OmNuh2*, *mass_split*, *T_CMB=None*)

Returns the neutrinos mass(es) for a given OmNuh2, according to the splitting convention specified by the user.

Parameters

- **OmNuh2** (*float*) – Neutrino energy density at z=0 times h^2
- **mass_split** (*str*) – indicates how the masses should be split up Should be one of 'normal', 'inverted', 'equal' or 'sum'.
- **T_CMB** (*float, optional*) – Temperature of the CMB (K). Default: 2.725.

Returns Neutrino mass(es) corresponding to this Omeganuh2

Return type float or array-like

pyccl.pk2d module

class pyccl.pk2d.**Pk2D**(*pkfunc=None*, *a_arr=None*, *lk_arr=None*, *pk_arr=None*, *is_logp=True*, *extrap_order_lok=1*, *extrap_order_hik=2*, *cosmo=None*)

Bases: object

A power spectrum class holding the information needed to reconstruct an arbitrary function of wavenumber and scale factor.

Parameters

- **(** (*pkfunc*) – obj:function): a function returning a floating point number or numpy array with the signature *f(k,a)*, where k is a wavenumber (in units of Mpc^-1) and a is the scale factor. The function must able to take numpy arrays as *k*. The function must return the value(s) of the power spectrum (or its natural logarithm, depending on the value of *is_logp*. The power spectrum units should be compatible with those used by CCL (e.g. if you're passing a matter power spectrum, its units should be Mpc^3). If this argument is not *None*, this function will be sampled at the values of k and a used internally by CCL to store the linear and non-linear power spectra.
- **a_arr** (*array*) – an array holding values of the scale factor
- **lk_arr** (*array*) – an array holding values of the natural logarithm of the wavenumber (in units of Mpc^-1).
- **pk_arr** (*array*) – a 2D array containing the values of the power spectrum at the values of the scale factor and the wavenumber held by *a_arr* and *lk_arr*. The shape of this array must be *[na,nk]*, where *na* is the size of *a_arr* and *nk* is the size of *lk_arr*. This array can be provided in a flattened form as long as the total size matches *nk*na*. The array can hold the values of the natural logarithm of the power spectrum, depending on the value of *is_logp*. If *pkfunc* is not None, then *a_arr*, *lk_arr* and *pk_arr* are ignored. However, either *pkfunc* or all of the last three array must be non-None. Note that, if you pass your own Pk array, you are responsible of making sure that it is sufficiently well sampled (i.e. the resolution of *a_arr* and *lk_arr* is high enough to sample the main features in the power spectrum). For reference, CCL will use bicubic interpolation to evaluate the power spectrum at any intermediate point in k and a.
- **extrap_order_lok** (*int*) – extrapolation order to be used on k-values below the minimum of the splines (use 0, 1 or 2). Note that the extrapolation will be done in either log(P(k)) or P(k), depending on the value of *is_logp*.
- **extrap_order_hik** (*int*) – extrapolation order to be used on k-values above the maximum of the splines (use 0, 1 or 2). Note that the extrapolation will be done in either log(P(k)) or P(k), depending on the value of *is_logp*.
- **is_logp** (*boolean*) – if True, pkfunc/pkarr return/hold the natural logarithm of the power spectrum. Otherwise, the true value of the power spectrum is expected. Note that arrays will be interpolated in log space if *is_logp* is set to *True*.
- **cosmo** (Cosmology) – Cosmology object. The cosmology object is needed in order if *pkfunc* is not *None*. The object is used to determine the sampling rate in scale factor and wavenumber.

eval(*k*, *a*, *cosmo*)

Evaluate power spectrum.

Parameters

- **k** (*float or array_like*) – wavenumber value(s) in units of Mpc^-1.
- **a** (*float*) – value of the scale factor
- **cosmo** (Cosmology) – Cosmology object. The cosmology object is needed in order to evaluate the power spectrum outside the interpolation range in *a*. E.g. if you want to evaluate the power spectrum at a very small a, not covered by the arrays you passed when initializing this object, the power spectrum will be extrapolated from the earliest available value using the linear growth factor (for which a cosmology is needed).
- **a_arr** (*array*) – an array holding values of the scale factor.

Returns value(s) of the power spectrum.

Return type float or array_like

pyccl.power module

pyccl.power.**linear_matter_power**(*cosmo*, *k*, *a*)
The linear matter power spectrum; Mpc^3.

Parameters

- **cosmo** (Cosmology) – Cosmological parameters.
- **k** (*float or array_like*) – Wavenumber; Mpc^-1.
- **a** (*float*) – Scale factor.

Returns Linear matter power spectrum; Mpc^3.

Return type float or array_like

pyccl.power.**nonlin_matter_power**(*cosmo*, *k*, *a*)
The nonlinear matter power spectrum; Mpc^3.

Parameters

- **cosmo** (Cosmology) – Cosmological parameters.
- **k** (*float or array_like*) – Wavenumber; Mpc^-1.
- **a** (*float*) – Scale factor.

Returns Nonlinear matter power spectrum; Mpc^3.

Return type float or array_like

pyccl.power.**sigma8**(*cosmo*)
RMS variance in a top-hat sphere of radius 8 Mpc/h.

Note: 8 Mpc/h is rescaled based on the Hubble constant.

Parameters **cosmo** (Cosmology) – Cosmological parameters.

Returns RMS variance in top-hat sphere of radius 8 Mpc/h.

Return type float

pyccl.power.**sigmaR**(*cosmo*, *R*, *a=1.0*)
RMS variance in a top-hat sphere of radius R in Mpc.

Parameters

- **cosmo** (Cosmology) – Cosmological parameters.
- **R** (*float or array_like*) – Radius; Mpc.
- **a** (*float*) – optional scale factor; defaults to a=1

Returns

RMS variance in the density field in top-hat sphere; Mpc.

Return type float or array_like

pyccl.power.**sigmaV**(*cosmo*, *R*, *a=1.0*)
RMS variance in the displacement field in a top-hat sphere of radius R. The linear displacement field is the gradient of the linear density field.

Parameters

- **cosmo** (Cosmology) – Cosmological parameters.
- **R** (*float or array_like*) – Radius; Mpc.
- **a** (*float*) – optional scale factor; defaults to a=1

Returns

RMS variance in the displacement field in top-hat sphere.

Return type sigmaV (float or array_like)

pyccl.pyutils module

Utility functions to analyze status and error messages passed from CCL, as well as wrappers to automatically vectorize functions.

pyccl.pyutils.**check**(*status*, *cosmo=None*)
Check the status returned by a ccllib function.

Parameters

- **status** (int or core.error_types) – Flag or error describing the success of a function.
- **cosmo** (Cosmology, optional) – A Cosmology object.

pyccl.pyutils.**debug_mode**(*debug*)
Toggle debug mode on or off. If debug mode is on, the C backend is forced to print error messages as soon as they are raised, even if the flow of the program continues. This makes it easier to track down errors.

If debug mode is off, the C code will not print errors, and the Python wrapper will raise the last error that was detected. If multiple errors were raised, all but the last will be overwritten within the C code, so the user will not necessarily be informed of the root cause of the error.

Parameters **debug** (*bool*) – Switch debug mode on (True) or off (False).

pyccl.tracers module

class pyccl.tracers.**CMBLensingTracer**(*cosmo*, *z_source*, *n_samples=100*)
Bases: *pyccl.tracers.Tracer*

A Tracer for CMB lensing.

Parameters

- **cosmo** (Cosmology) – Cosmology object.
- **z_source** (*float*) – Redshift of source plane for CMB lensing.
- **nsamples** (*int, optional*) – number of samples over which the kernel is desired. These will be equi-spaced in radial distance. The kernel is quite smooth, so usually O(100) samples is enough.

class pyccl.tracers.**NumberCountsTracer**(*cosmo*, *has_rsd*, *dndz*, *bias*, *mag_bias=None*)
Bases: *pyccl.tracers.Tracer*

Specific *Tracer* associated to galaxy clustering with linear scale-independent bias, including redshift-space distortions and magnification.

Parameters

- **cosmo** (Cosmology) – Cosmology object.
- **has_rsd** (*bool*) – Flag for whether the tracer has a redshift-space distortion term.
- **dndz** (*tuple of arrays*) – A tuple of arrays (z, N(z)) giving the redshift distribution of the objects. The units are arbitrary; N(z) will be normalized to unity.
- **bias** (*tuple of arrays*) – A tuple of arrays (z, b(z)) giving the galaxy bias. If *None*, this tracer won't include a term proportional to the matter density contrast.
- **mag_bias** (*tuple of arrays, optional*) – A tuple of arrays (z, s(z)) giving the magnification bias as a function of redshift. If *None*, the tracer is assumed to not have magnification bias terms. Defaults to None.

class pyccl.tracers.**Tracer**
Bases: object

Tracers contain the information necessary to describe the contribution of a given sky observable to its cross-power spectrum with any other tracer. Tracers are composed of 4 main ingredients:

- A radial kernel: this expresses the support in redshift/distance over which this tracer extends.
- A transfer function: this is a function of wavenumber and scale factor that describes the connection between the tracer and the power spectrum on different scales and at different cosmic times.
- An ell-dependent prefactor: normally associated with angular derivatives of a given fundamental quantity.
- The order of the derivative of the Bessel functions with which they enter the computation of the angular power spectrum.

A *Tracer* object will in reality be a list of different such tracers that get combined linearly when computing power spectra. Further details can be found in Section 4.9 of the CCL note.

add_tracer(*cosmo*, *kernel=None*, *transfer_ka=None*, *transfer_k=None*, *transfer_a=None*, *der_bessel=0*, *der_angles=0*, *is_logt=False*, *extrap_order_lok=0*, *extrap_order_hik=2*)
Adds one more tracer to the list contained in this *Tracer*.

Parameters

- **cosmo** (Cosmology) – cosmology object.
- **kernel** (*tulple of arrays, optional*) – A tuple of arrays (*chi*, *w_chi*) describing the radial kernel of this tracer. *chi* should contain values of the comoving radial distance in increasing order, and *w_chi* should contain the values of the kernel at those values of the radial distance. The kernel will be assumed to be zero outside the range of distances covered by *chi*. If *kernel* is *None* a constant kernel w(chi)=1 will be assumed everywhere.
- **transfer_ka** (*tuple of arrays, optional*) – a tuple of arrays (*a*,'lk','t_ka') describing the most general transfer function for a tracer. *a* should be an array of scale factor values in increasing order. *lk* should be an array of values of the natural logarithm of the wave number (in units of inverse Mpc) in increasing order. *t_ka* should be an array of shape *(na,nk)*, where *na* and *nk* are the sizes of *a* and *lk* respectively. *t_ka* should hold the values of the transfer function at the corresponding values of *a* and *lk*. If your transfer function is factorizable (i.e. T(a,k) = A(a) * K(k)), it is more efficient to set this to *None* and use *transfer_k* and *transfer_a* to describe K and A respectively. The transfer function will be assumed continuous and constant outside the range of scale factors covered by *a*.

It will be extrapolated using polynomials of order *extrap_order_lok* and *extrap_order_hik* below and above the range of wavenumbers covered by *lk* respectively. If this argument is not *None*, the values of *transfer_k* and *transfer_a* will be ignored.

- **transfer_k** (*tuple of arrays, optional*) – a tuple of arrays (*lk*,'t_k') describing the scale-dependent part of a factorizable transfer function. *lk* should be an array of values of the natural logarithm of the wave number (in units of inverse Mpc) in increasing order. *t_k ' should be an array of the same size holding the values of the k-dependent part of the transfer function at those wavenumbers. It will be extrapolated using polynomials of order 'extrap_order_lok* and *extrap_order_hik* below and above the range of wavenumbers covered by *lk* respectively. If *None*, the k-dependent part of the transfer function will be set to 1 everywhere.
- **transfer_a** (*tuple of arrays, optional*) – a tuple of arrays (*a*,'t_a') describing the time-dependent part of a factorizable transfer function. *a* should be an array of scale factor values in increasing order. *t_a* should contain the time-dependent part of the transfer function at those values of the scale factor. The time dependence will be assumed continuous and constant outside the range covered by *a*. If *None*, the time-dependent part of the transfer function will be set to 1 everywhere.
- **der_bessel** (*int*) – order of the derivative of the Bessel functions with which this tracer enters the calculation of the power spectrum. Allowed values are -1, 0, 1 and 2. 0, 1 and 2 correspond to the raw functions, their first derivatives or their second derivatives. -1 corresponds to the raw functions divided by the square of their argument. We enable this special value because this type of dependence is ubiquitous for many common tracers (lensing, IAs), and makes the corresponding transfer functions more stables for small k or chi.
- **der_angles** (*int*) – integer describing the ell-dependent prefactor associated with this tracer. Allowed values are 0, 1 and 2. 0 means no prefactor. 1 means a prefactor ell*(ell+1), associated with the angular laplacian and used e.g. for lensing convergence and magnification. 2 means a prefactor sqrt((ell+2)!/(ell-2)!), associated with the angular derivatives of spin-2 fields (e.g. cosmic shear, IAs).
- **is_logt** (*bool*) – if *True*, *transfer_ka*, *transfer_k* and *transfer_a* will contain the natural logarithm of the transfer function (or their factorizable parts). Default is *False*.
- **extrap_order_lok** (*int*) – polynomial order used to extrapolate the transfer functions for low wavenumbers not covered by the input arrays.
- **extrap_order_hik** (*int*) – polynomial order used to extrapolate the transfer functions for high wavenumbers not covered by the input arrays.

class pyccl.tracers.**WeakLensingTracer**(*cosmo*, *dndz*, *has_shear=True*, *ia_bias=None*)

Bases: *pyccl.tracers.Tracer*

Specific *Tracer* associated to galaxy shape distortions including lensing shear and intrinsic alignments within the L-NLA model.

Parameters

- **cosmo** (Cosmology) – Cosmology object.
- **dndz** (*tuple of arrays*) – A tuple of arrays (z, N(z)) giving the redshift distribution of the objects. The units are arbitrary; N(z) will be normalized to unity.
- **has_shear** (*bool*) – set to *False* if you want to omit the lensing shear contribution from this tracer.
- **ia_bias** (*tuple of arrays, optional*) – A tuple of arrays (z, A_IA(z)) giving the intrinsic alignment amplitude A_IA(z). If *None*, the tracer is assumped to not have

intrinsic alignments. Defaults to None.

pyccl.tracers.**get_density_kernel**(*cosmo*, *dndz*)

This convenience function returns the radial kernel for galaxy-clustering-like tracers. Given an unnormalized redshift distribution, it returns two arrays: chi, w(chi), where chi is an array of radial distances in units of Mpc and w(chi) = p(z) * H(z), where H(z) is the expansion rate in units of Mpc^-1 and p(z) is the normalized redshift distribution.

Parameters

- **cosmo** (Cosmology) – cosmology object used to transform redshifts into distances.
- **dndz** (*tulple of arrays*) – A tuple of arrays (z, N(z)) giving the redshift distribution of the objects. The units are arbitrary; N(z) will be normalized to unity.

pyccl.tracers.**get_kappa_kernel**(*cosmo*, *z_source*, *nsamples*)

This convenience function returns the radial kernel for CMB-lensing-like tracers.

Parameters

- **cosmo** (Cosmology) – Cosmology object.
- **z_source** (*float*) – Redshift of source plane for CMB lensing.
- **nsamples** (*int*) – number of samples over which the kernel is desired. These will be equi-spaced in radial distance. The kernel is quite smooth, so usually O(100) samples is enough.

pyccl.tracers.**get_lensing_kernel**(*cosmo*, *dndz*, *mag_bias=None*)

This convenience function returns the radial kernel for weak-lensing-like. Given an unnormalized redshift distribution and an optional magnification bias function, it returns two arrays: chi, w(chi), where chi is an array of radial distances in units of Mpc and w(chi) is the lensing shear kernel (or the magnification one if *mag_bias* is not *None*).

Parameters

- **cosmo** (Cosmology) – cosmology object used to transform redshifts into distances.
- **dndz** (*tulple of arrays*) – A tuple of arrays (z, N(z)) giving the redshift distribution of the objects. The units are arbitrary; N(z) will be normalized to unity.
- **mag_bias** (*tuple of arrays, optional*) – A tuple of arrays (z, s(z)) giving the magnification bias as a function of redshift. If *None*, s=0 will be assumed

14.1.2 Module contents

CHAPTER 15

Changelog

15.1 v 2.0 Changes

15.1.1 Python library

- Made CAMB the default Boltzmann code (#685).
- Added check to ensure the number of relativistic neutrinos is positive (#684).
- Added massive neutrinos to Omega_m (#680).
- Chamged neutrino API options to be intuitive and consistent (#681).
- Enabled CAMB (#677)
- Fixed bug when normalizing the linear power spectrum using sigma8 and the mu-Sigma modified gravity model (#677).
- Enabled the mu-Sigma modified gravity model for any linear power spectrum (#677).
- Refactored the power spectrum normalization routines to only run CAMB/CLASS once when using sigma8 (#677).
- Fixed a bug where the power spectrum normalization was not being set accurately when using sigma8 (#677).
- Added warnings for inconsistent models (#676).
- Moved CLASS interface to python (#652).
- Moved all benchmarks and tests to python (#653).
- Changed IA bias normalization to be consistent with A_IA=1 (#630).
- Implemented generalized models for tracers (#630)
- Improved error reporting for `angular_cl` computations (#567).
- Deprecated the `pyccl.redshifts` module (#579).

- Remove global splines for RSD correlation functions. These are now stored per cosmology. Further, they are now rebuilt on-the-fly for a given cosmology if a new scale factor is requested. (#582)
- Allow spline, numerical and constant parameters to be set from Python (#557).
- Deprecated transfer function options 'ccl_emulator', 'ccl_fitting_function' 'ccl_boltzmann', 'ccl_boltzmann_class' and 'ccl_boltzmann_camb' (#610). These were either not implemented or aliases for another option.
- Renamed transfer function option 'ccl_none' to 'ccl_transfer_none' to avoid ambiguity (#610).
- Refactored transfer function and matter power spectrum options to allow any combination, even unphysical ones (#610).
- Added capability to use the halo model power spectrum as the primary non-linear power spectrum in the code (#610).
- Fixed infinite loop bug in splitting sum of neutrino masses into individual masses (#605).
- Added custom Halofit code (#611).
- Added `has_density` and `has_shear` tags to `Tracer` constructors.
- Changed TCMB to T_CMB everywhere (#615)
- Added support for modified gravity via mu / Sigma (scale-independent) parameterisation (#442)

15.1.2 C library

- Added massive neutrinos to Omega_m (#680).
- Moved CLASS interface to python (#652).
- Added OpenMP (#651).
- Removed all benchmarks from C and kept only the C unit tests in C (#653).
- Implemented generalized models for tracers (#630)
- Fixed memory leak in CLASS power spectrum computations (#561, #562).
- Fixed a bug where CLASS would crash due to small rounding errors at z = 0 when evaluating power spectra (#563, #564).
- Fixed bug in fftlog for some complex arguments (#565, #566).
- Replaced custom gamma function with that from GSL (#570).
- Deprecated the `ccl_redshifts.h` functions (#579).
- Refactored spline and numerical parameters to be allocated per cosmology (#557).
- Allow global physical constants to be changed (#557).
- Fixed memory leaks in `ccl_correlation.c` (#581).
- Deprecated transfer function options 'ccl_emulator', 'ccl_fitting_function' 'ccl_boltzmann', 'ccl_boltzmann_class' and 'ccl_boltzmann_camb' (#610). These were either not implemented or aliases for another option.
- Renamed transfer function option 'ccl_none' to 'ccl_transfer_none' to avoid ambiguity (#610).
- Refactored transfer function and matter power spectrum options to allow any combination, even unphysical ones (#610).
- Added additional header and source files for clarity (#610).

- Added capability to use the halo model power spectrum as the primary non-linear power spectrum in the code (#610).
- Fixed infinite loop bug in splitting sum of neutrino masses into individual masses (#605).
- Added custom Halofit code (#611).
- Separated Limber and Non-Limber C_ell calculations (#614)
- Added `has_density` and `has_shear` flags to ClTracers (#614)
- Simplified C_ell unit tests (#614)
- Changed TCMB to T_CMB everywhere (#615)
- Fixed a small bug in the w_tophat expression and increased precision (#607)
- Deprecated the use of GSL spline accelerators (#626)
- Added support for modified gravity via mu / Sigma (scale-independent) parameterisation (#442)

15.2 v 1.0 API changes :

15.2.1 C library

- Deprecated the `native` non-Limber angular power spectrum method (#506).
- Renamed `ccl_lsst_specs.c` to `ccl_redshifts.c`, deprecated LSST-specific redshift distribution functionality, introduced user-defined true dNdz (changes in call signature of `ccl_dNdz_tomog`). (#528).

15.2.2 Python library

- Renamed `lsst_specs.py` to `redshifts.py`, deprecated LSST-specific redshift distribution functionality, introduced user-defined true dNdz (changes in call signature of `dNdz_tomog`). (#528).
- Deprecated the `native` non-Limber angular power spectrum method (#506).
- Deprecated the `Parameters` object in favor of only the `Cosmology` object (#493).
- Renamed the `ClTracer` family of objects (#496).
- Various function parameter name changes and documentation improvements (#464).

15.3 v 0.4 API changes:

Summary: added halo model matter power spectrum calculation and halo mass-concentration relations. Change to sigma(R) function so that it now has time dependence: it is now sigma(R,a). Added a sigmaV(R,a) function, where sigmaV(R,a) is the variance in the displacement field smoothed on scale R at scale-factor a.

15.3.1 C library

In ccl_halomod.c:

Added this source file. Contains functions to compute halo-model matter power spectra and also mass-concentration relations.

In ccl_power.c

Added sigmaV, changed sigma(R) -> sigma(R,a)

In ccl_massfunc.c

Added Sheth & Tormen (1999) mass function.

15.3.2 Python library

sigmaR(R) defaults to sigmaR(R,a=1) unless a is specified. sigmaV(R) is also added. New functions ccl.halomodel_matter_power and ccl.halo_concentration.

15.4 v 0.3 API changes:

Summary: the user interface for setting up cosmologies with neutrinos has been altered. Users should from now on pass Neff, the effective number of relativistic neutrino species in the early universe, and mnu, either a sum or neutrino masses or a set of 3 neutrinos masses.

15.4.1 C library

In ccl_core.c:

In the function, 'ccl_parameters_create', the arguements 'double N_nu_rel', and 'double N_nu_mass' have been removed. The arguments 'double Neff' and 'ccl_mnu_convention mnu_type' have been added. The argument 'mnu' has changed in type from 'double mnu' to 'double* mnu'.

Similar changes apply in 'ccl_cosmology_create_with_params' and all 'ccl_parameters_create...nu' convenience functions.

Additionally, in the function 'ccl_parameters_create' and 'ccl_cosmology_create_with_params', arguments have been added for the parameters of the BCM baryon model; these are 'double bcm_log10Mc', 'double bcm_etab', and 'double bcm_ks'.

In ccl_neutrinos.c:

The function ccl_Omeganuh2_to_Mnu has been renamed ccl_nu_masses. The arguments 'double a' and 'gsl_interp_accel* accel' have been removed. The argument 'ccl_neutrino_mass_splits mass_split' has been added.

15.4.2 Python wrapper

In core.py:

In the Parameters class, the arguments 'N_nu_rel', and 'N_nu_mass' have been removed. The optional arguments 'Neff', 'mnu_type', 'bcm_log10Mc', 'bcm_etab', and 'bcm_ks' have been added. Similar changes occur in the Cosmology class.

In neutrinos.py:

In the function Omeganuh2, the argument 'Neff' has been removed. It is now fixed to the length of the argument 'mnu'. The function 'Omeganuh2_to_Mnu' has been renamed 'nu_masses'. The arguments 'a' and 'Neff' have been removed. The argument 'mass_split' has been added. The argument 'TCMB' has been changed to 'T_CMB'.

15.4.3 Other changes since release 0.2.1 (September 2017):

CLASS is no longer included as part of CCL; it can instead of easily downloaded via the class_install.py script and this procedure is documented.

Tutorial added for usage with MCMC

Added support for BCM baryon model

cpp compatibility improved

Python 3 support added

Added support for computing the nonlinear matter power spectrum with CosmicEmu

Added support for CMB lensing observables, including splines for cosmological quantities to higher redshift

Added the ability to return useful functions such as dNdz for a tracer Cl object.

Clarified license

Values of the physical constants have changed to CODATA2014/IAU2015

Python Module Index

p

Index

L

M

N

O

P

R

S

T

W