
Core Cosmology Library Documentation

Release 1.0

LSST DESC

Feb 15, 2019

1	Installation	3
2	Installation for developers	7
3	Reporting a bug	9
4	pyccl	11
5	Doxygen C Library Documentation	25
6	Citing CCL	27
	Python Module Index	29

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
- Correlation functions C_ℓ for arbitrary combinations of tracers including galaxies, shear and number counts
- Halo mass function dn/dM and halo bias $b(M)$
- Approximate baryonic modifications to the matter power spectra $\Delta_{\text{baryons}}^2$
- Simple modified gravity extensions $\Delta f(z)$

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

1.1 With pip

CCL is available as a Python package through PyPi. To install, simply run:

```
$ pip install pycc1
```

This should work as long as CMake is installed on your system. Once installed, take it for a spin in by following some example notebooks [here](#).

If you want the C library available, you have to follow the manual installation.

1.2 Dependencies and Requirements

You can also install CCL from the source, however in order to compile CCL you need a few libraries:

- GNU Scientific Library [GSL](#), version 2.1 or above
- [FFTW3](#) version 3.1 or above
- [CLASS](#) version 2.6.3 or above
- [FFTlog](#) ([here](#) and [here](#)) is provided within CCL, with minor modifications.

Additionally, to build the code you will need

- [CMake](#) version 3.2 or above.
- [SWIG](#) is needed if you wish to modify CCL and have it available in Python.

CMake is the only requirement that needs to be installed manually if you are using pip.

On Ubuntu:

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

On MacOS X you can either install with a DMG from [this page](#) or with a package manager such as brew, MacPorts, or Fink. For instance with brew:

```
$ brew install cmake
```

You will avoid potential issues if you install `GSL` and `FFTW` on your system before building CCL, but is only necessary if you want to properly install the C library. Otherwise CMake will automatically download and build the missing requirements in order to compile CCL.

To install all the dependencies at once, and avoid having CMake recompiling them, for instance on Ubuntu:

```
$ sudo apt-get install cmake swig libgsl-dev libfftw3-dev
```

1.3 Compile and install the CCL C library

To download the latest version of CCL:

```
$ git clone https://github.com/LSSTDESC/CCL.git
$ cd CCL
```

or download and extract the latest stable release from [here](#). Then, from the base CCL directory run:

```
$ mkdir build && cd build
$ cmake ..
```

This will run the configuration script, try to detect the required dependencies on your machine and generate a Makefile. Once CMake has been configured, to build and install the library simply run for the `build` directory:

```
$ make
$ make install
```

Often admin privileges will be needed to install the library. If you have those just type:

```
$ sudo make install
```

Note: This is the default install procedure, but depending on your system you might want to customize the install process. Here are a few common configuration options:

In case you have several C compilers, you can direct which one for CMake to use by setting the environment variable `CC` **before** running CMake:

```
$ export CC=gcc
```

By default, CMake will try to install CCL in `/usr/local`. If you would like to instead install elsewhere (such as if you don't have admin privileges), you can specify it **before** running CMake by doing:

```
$ cmake -DCMAKE_INSTALL_PREFIX=/path/to/install ..
```

This will instruct CMake to install CCL in the following folders: `/path/to/install/include`, `code:/path/to/install/share`, and `/path/to/install/lib`.

Depending on where you install CCL you might need to add the installation path to your `PATH` and `LD_LIBRARY_PATH` environment variables. In the default case, this is accomplished with:

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


To make sure that everything is working properly, you can run all unit tests after installation by running from the root CCL directory:

```
$ check_ccl
```

Assuming that the tests pass, you have successfully installed CCL!

If you ever need to uninstall CCL, run the following from the `build` directory:

```
$ make uninstall
```

You may need to prepend a `sudo` if you installed CCL in a protected folder.

Once the CLASS library is installed, CCL can be easily installed using an *autotools*-generated configuration file. To install CCL, from the base directory (the one where this file is located) run:

Often admin privileges will be needed to install the library. If you have those just type:

```
sudo make install
```

If you don't have admin privileges, you can still install the library in a user-defined directory by running

```
./configure --prefix=/path/to/install  
make  
make install
```

where `/path/to/install` is the absolute path to the directory where you want the library to be installed. If non-existing, this will create two directories, `/path/to/install/include` and `/path/to/install/lib`, and the library and header files will be installed there. Note that, in order to use CCL with your own scripts you'll have to add `/path/to/install/lib` to your `LD_LIBRARY_PATH`. CCL has been successfully installed on several different Linux and Mac OS X systems.

To make sure that everything is working properly, you can run all unit tests after installation by running

```
make check
```

Assuming that the tests pass, you can then move on to installing the Python wrapper (optional).

After pulling a new version of CCL from the [GitHub repository](#), you can recompile the library by running:

```
make clean; make uninstall  
make  
make install
```

1.4 Install the pyccl Python module

CCL also comes with a Python wrapper, called `pyccl`, which can be built and installed regardless of whether you install the C library. For convenience, we provide a PyPi hosted package which can be installed simply by running:

```
$ pip install pyccl # append --user for single user install
```

This only assumes that `CMake` is available on your system, you don't need to download the source yourself.

You can also build and install `pyccl` from the CCL source, again without necessarily installing the C library. Download the latest version of CCL:

```
$ git clone https://github.com/LSSTDESC/CCL.git
$ cd CCL
```

And from the root CCL folder, simply run:

```
$ python setup.py install # append --user for single user install
```

The `pyccl` module will be installed into a sensible location in your `PYTHONPATH`, and so should be picked up automatically by your Python interpreter. You can then simply import the module using `import pyccl`.

You can quickly check whether `pyccl` has been installed correctly by running `python -c "import pyccl"` and checking that no errors are returned.

For a more in-depth test to make sure everything is working, run from the root CCL directory:

```
python setup.py test
```

This will run the embedded unit tests (may take a few minutes).

Whatever the install method, if you have `pip` installed, you can always uninstall the python wrapper by running:

```
pip uninstall pyccl
```

For quick introduction to *CCL* in Python look at notebooks in `**_tests/_**`.

1.5 Compiling against an external version of CLASS

The default installation procedure for CCL implies automatically downloading and installing a tagged version of CLASS. Optionally, you can also link CCL against a different version of CLASS. This is useful if you want to use a modified version of CLASS, or a different or more up-to-date version of the standard CLASS.

To compile CCL with an external version of CLASS, just run the following `CMake` command at the first configuration step of the install (from the build directory, make sure it is empty to get a clean configuration):

```
$ cmake -DEXTERNAL_CLASS_PATH=/path/to/class ..
```

the rest of the build process should be the same.

1.6 Docker image installation

The Dockerfile to generate a Docker image is included in the CCL repository as `Dockerfile`. This can be used to create an image that Docker can spool up as a virtual machine, allowing you to utilize CCL on any infrastructure with minimal hassle. The details of Docker and the installation process can be found at [this link](#). Once Docker is installed, it is a simple process to create an image! In a terminal of your choosing (with Docker running), type the command `docker build -t ccl .` in the CCL directory.

The resulting Docker image has two primary functionalities. The first is a CMD that will open Jupyter notebook tied to a port on your local machine. This can be used with the following run command: `docker run -p 8888:8888 ccl`. You can then access the notebook in the browser of your choice at `localhost:8888`. The second is to access the bash itself, which can be done using `docker run -it ccl bash`.

This Dockerfile currently contains all installed C libraries and the Python wrapper. It currently uses `continuumio/anaconda` as the base image and supports `ipython` and Jupyter notebook. There should be minimal slowdown due to the virtualization.

Installation for developers

2.1 Development workflow

Installing CCL on the system is not a good idea when doing development, you can compile and run all the libraries and examples directly from your local copy. The only subtlety when not actually installing the library is that one needs to define the environment variable `CCL_PARAM_FILE` pointing to `include/ccl_params.ini`:

```
$ export CCL_PARAM_FILE=/path/to/your/ccl/folder/include/ccl_params.ini
```

Failure to define this environment variable will result in violent segmentation faults!

2.2 Working on the C library

Here are a few common steps when working on the C library:

Cloning a local copy and CCL and compiling it:

```
$ git clone https://github.com/LSSTDESC/CCL
$ mkdir -p CCL/build && cd CCL/build
$ cmake ..
$ make
```

Updating local copy from master, recompiling what needs recompiling, and running the test suite:

```
$ git pull      # From root folder
$ make -Cbuild # The -C option allows you to run make from a different directory
$ build/check_ccl
```

Compiling (or recompiling) an example in the `CCL/examples` folder:

```
$ cd examples # From root folder
$ make -C../build ccl_sample_pkemu
$ ./ccl_sample_pkemu
```

Reconfiguring from scratch (in case something goes terribly wrong):

```
$ cd build
$ rm -rf *
$ cmake ..
$ make
```

2.3 Working on the Python library

Here are a few common steps when working on the Python module:

Building the python module:

```
$ python setup.py build
```

After that, you can start your interpreter from the root CCL folder and import pyccl.

Running the tests after a modification of the C library:

```
$ python setup.py build
$ python setup.py test
```

CHAPTER 3

Reporting a bug

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.

4.1 pyccl package

4.1.1 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.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 comoving angular distance is defined such that the comoving distance between two objects at a fixed scale factor separated by an angle η is $\eta D_T(a)$ 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 $5 * \log(\text{luminosity distance} / 10 \text{ pc})$.

Note: The distance modulus can be used to convert between apparent and absolute magnitudes via $m = M + \text{distance modulus}$, 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, $d\ln D/d\ln a$.

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 $H(a)/H_0$.

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.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. Available: ‘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

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. Available: ‘matter’: cold dark matter 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.cls module

class `pyccl.cls.CMBLensingTracer` (*cosmo*, *z_source*)

Bases: `pyccl.cls.Tracer`

A Tracer for CMB lensing.

Parameters

- **cosmo** (*Cosmology*) – Cosmology object.
- **z_source** (*float*) – Redshift of source plane for CMB lensing.

class `pyccl.cls.NumberCountsTracer` (*cosmo*, *has_rsd*, *dndz*, *bias*, *mag_bias=None*)

Bases: `pyccl.cls.Tracer`

A Tracer for galaxy number counts (galaxy clustering).

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.
- **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.cls.Tracer` (**args*, ***kwargs*)

Bases: `object`

A tracer of the matter density field.

Note: This class cannot be used directly. Use one of *NumberCountsTracer*, *WeakLensingTracer* or *CMBLensingTracer* instead.

This class contains all information describing the transfer function of a tracer (e.g., galaxy density, lensing shear) of the matter distribution.

get_internal_function (*cosmo, function, a*)

Method to evaluate any internal function of redshift for this tracer.

Parameters

- **cosmo** (*Cosmology*) – Cosmology object.
- **function** (*str*) – Specifies which function to evaluate. Must be one of ‘dndz’: number density ‘bias’: bias ‘mag_bias’: magnification bias ‘red_frac’: red fraction ‘ia_bias’: intrinsic alignment bias ‘lensing_win’: weak lensing window function ‘mag_win’: magnification window function
- (*a*) – obj: float or array-like): list of scale factors at which to evaluate the function.

Returns Array of function values at the input scale factors.

class `pyccl.cls.WeakLensingTracer` (*cosmo, dndz, ia_bias=None, red_frac=None*)

Bases: `pyccl.cls.Tracer`

A Tracer for weak lensing shear (galaxy shapes).

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.
- **ia_bias** (*tuple of arrays, optional*) – A tuple of arrays (*z*, *b_IA(z)*) giving the intrinsic alignment amplitude *b_IA(z)*. If *None*, the tracer is assumed to not have intrinsic alignments. Defaults to *None*.
- **red_frac** (*tuple of arrays, optional*) – A tuple of arrays (*z*, *f_red(z)*) giving the red fraction of galaxies as a function of redshift. If *None*, then the tracer is assumed to not have a red fraction. Defaults to *None*.

`pyccl.cls.angular_cl` (*cosmo, cltracer1, cltracer2, ell, l_limber=-1.0, l_logstep=1.05, l_linstep=20.0*)

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.
- **l_limber** (*float*) – Angular wavenumber beyond which Limber’s approximation will be used. Defaults to -1.
- **l_logstep** (*float*) – logarithmic step in *ell* at low multipoles. Defaults to 1.05.
- **l_linstep** (*float*) – linear step in *ell* at high multipoles. Defaults to 20.

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.constants module

This file exposes constants present in CCL.

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.

The classes in this module accept strings indicating which model to use for various physical quantities (e.g., the transfer function). The various options are as follows.

transfer_function options

- ‘emulator’: the transfer function defined by the Comsic Emu
- ‘fitting_function’: the Eisenstein and Hu (1998) fitting function
- ‘eisenstein_hu’: the Eisenstein and Hu (1998) fitting function
- ‘bbks’: the BBKS approximation
- ‘boltzmann’: use CLASS to compute the transfer function
- ‘boltzmann_class’: use CLASS to compute the transfer function
- ‘class’: use CLASS to compute the transfer function
- ‘boltzmann_camb’: not implemented
- ‘camb’: not implemented

matter_power_spectrum options

- ‘halo_model’: use a halo model
- ‘halofit’: use HALOFIT
- ‘linear’: neglect non-linear power spectrum contributions
- ‘emu’: use the Cosmic Emu

baryons_power_spectrum options

- ‘nobaryons’: neglect baryonic contributions to the power spectrum
- ‘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
- ‘watson’: the Watson et al. mass function
- ‘angulo’: the Angulo et al. mass function
- ‘shethormen’: the Sheth and Tormen mass function

halo_concentration options

- ‘bhattacharya2011’: Bhattacharya et al. (2011) relation
- ‘duffy2008’: Duffy et al. (2008) relation
- ‘constant_concentration’: use a constant concentration

mnu_type options This parameter specifies the model for massive neutrinos.

- ‘list’: specify each mass yourself in eV
- ‘sum’: use the normal hierarchy to convert total mass to individual masses
- ‘sum_inverted’: use the inverted hierarchy to convert total mass to individual masses
- ‘sum_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
- ‘equalize’: redistribute the total mass equally before using the Cosmic Emu. This option may result in slight internal inconsistencies in the physical model assumed for neutrinos.

```
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, mnu_type=None, w0=-1.0, wa=0.0,
bcm_log10Mc=14.079181246047625, bcm_etab=0.5,
bcm_ks=55.0, z_mg=None, df_mg=None, transfer_function='boltzmann_class', 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.

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.
- **mnu_type** (*str*, optional) – The type of massive neutrinos.
- **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.
- **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.
- **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_class`’.
- **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 `mnu_type = ‘sum_equal’`, and ‘`equalize`’, which will redistribute masses to be equal right before calling the emulator but results in internal inconsistencies. Defaults to ‘`strict`’.

compute_distances ()

 Compute the distance splines.

compute_growth ()

 Compute the growth function.

compute_power ()

Compute the power spectrum.

has_distances ()

Checks if the distances have been precomputed.

Returns True if precomputed, False otherwise.

Return type bool

has_growth ()

Checks if the growth function has been precomputed.

Returns True if precomputed, False otherwise.

Return type bool

has_power ()

Checks if the power spectra have been precomputed.

Returns True if precomputed, False otherwise.

Return type bool

has_sigma ()

Checks if sigma8 has been computed.

Returns True if precomputed, False otherwise.

Return type bool

classmethod read_yaml (*filename*)

Read the parameters from a YAML file.

Parameters *filename* (*str*) –

status ()

Get error status of the ccl_cosmology object.

Note: error statuses are currently under development.

Returns *str* containing the status message.

write_yaml (*filename*)

Write a YAML representation of the parameters to file.

Parameters *filename* (*str*) –

`pyccl.core.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.

pyccl.correlation module

Correlation function 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='fftllog'*)

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), ‘FFTLLog’ (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.

Args: `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.

Args: `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.

`pyccl.correlation.correlation_pi_sigma` (*cosmo, a, beta, pie, sig, use_spline=True*)

Compute the 3DRsd correlation in pi-sigma space.

Args: *cosmo* (Cosmology): A Cosmology object. *a* (float): scale factor. *pie* (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.correlation.correlation_spline_free` ()

Clear the global splines created from if 'use_spline' was set to True.

pyccl.lsst_specs module**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 ρ_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, mnu, TCMB=2.725*)

Calculate $\Omega_{\nu} * h^2$ at a given scale factor given the sum of the neutrino masses.

Note: for all practical purposes, N_{eff} is simply $N_{\nu_{\text{mass}}}$.

Parameters

- **a** (*float*) – Scale factor, normalized to 1 today.
- **mnu** (*float or array_like*) – Neutrino mass (in eV)
- **TCMB** (*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, TCMB=2.725*)

Returns the neutrinos mass(es) for a given $\Omega_{\text{M}} h^2$, according to the splitting convention specified by the user.

Parameters

- **OmNuh2** (*float*) – Neutrino energy density at $z=0$ times h^2
- **mass_split** – indicates how the masses should be split up
- **TCMB** (*float, optional*) – Temperature of the CMB (K). Default: 2.725.

Returns Neutrino mass(es) corresponding to this $\Omega_{\nu h^2}$

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.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).

4.1.2 Module contents

The `pyccl` package contains all of the submodules that are implemented in individual files in CCL.

Doxygen C Library Documentation

CCL has basic [Doxygen](#) documentation for its C routines. This can be found in the directory `doc/html` within the CCL repository by opening the `index.html` file in your browser.

This document contains basic information about used structures and functions. At the end of document is provided code which implements these basic functions (also in `examples/ccl_sample_run.c`). More information about CCL functions and implementation can be found in `doc/0000-ccl_note/0000-ccl_note.pdf`.

The python routines are documented in situ; you can view the documentation for a function by calling `help(function name)` from within the Python environment.

CHAPTER 6

Citing CCL

The CCL is still under development and should be considered research in progress. You are welcome to re-use the code, which is open source and available under terms consistent with [BSD 3-Clause](#) licensing. If you make use of any of the ideas or software in this package in your own research, please cite them as “(LSST DESC, in preparation)” and provide a link to this repository: <https://github.com/LSSTDESC/CCL>. 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>. If you have comments, questions, or feedback, please [write us an issue](#). Finally, CCL uses code from the [FFTLog](#) package. We have obtained permission from the FFTLog author to include modified versions of his source code.

p

pyccl, 24
pyccl.background, 11
pyccl.cls, 14
pyccl.constants, 16
pyccl.core, 16
pyccl.correlation, 19
pyccl.massfunction, 21
pyccl.neutrinos, 22
pyccl.power, 23
pyccl.pyutils, 24

A

angular_cl() (in module pycccl.cls), 15

C

check() (in module pycccl.core), 19
 CMBLensingTracer (class in pycccl.cls), 14
 comoving_angular_distance() (in module pycccl.background), 11
 comoving_radial_distance() (in module pycccl.background), 12
 compute_distances() (pycccl.core.Cosmology method), 18
 compute_growth() (pycccl.core.Cosmology method), 18
 compute_power() (pycccl.core.Cosmology method), 18
 correlation() (in module pycccl.correlation), 19
 correlation_3d() (in module pycccl.correlation), 20
 correlation_3dRsd() (in module pycccl.correlation), 20
 correlation_3dRsd_avgmu() (in module pycccl.correlation), 20
 correlation_multipole() (in module pycccl.correlation), 21
 correlation_pi_sigma() (in module pycccl.correlation), 21
 correlation_spline_free() (in module pycccl.correlation), 21
 Cosmology (class in pycccl.core), 17

D

debug_mode() (in module pycccl.pyutils), 24
 distance_modulus() (in module pycccl.background), 12

G

get_internal_function() (pycccl.cls.Tracer method), 15
 growth_factor() (in module pycccl.background), 12
 growth_factor_unnorm() (in module pycccl.background), 12
 growth_rate() (in module pycccl.background), 12

H

h_over_h0() (in module pycccl.background), 13
 halo_bias() (in module pycccl.massfunction), 21
 has_distances() (pycccl.core.Cosmology method), 19

has_growth() (pycccl.core.Cosmology method), 19
 has_power() (pycccl.core.Cosmology method), 19
 has_sigma() (pycccl.core.Cosmology method), 19

L

linear_matter_power() (in module pycccl.power), 23
 luminosity_distance() (in module pycccl.background), 13

M

massfunc() (in module pycccl.massfunction), 21
 massfunc_m2r() (in module pycccl.massfunction), 21

N

nonlin_matter_power() (in module pycccl.power), 23
 nu_masses() (in module pycccl.neutrinos), 22
 NumberCountsTracer (class in pycccl.cls), 14

O

omega_x() (in module pycccl.background), 13
 Omeganuh2() (in module pycccl.neutrinos), 22

P

pycccl (module), 24
 pycccl.background (module), 11
 pycccl.cls (module), 14
 pycccl.constants (module), 16
 pycccl.core (module), 16
 pycccl.correlation (module), 19
 pycccl.massfunction (module), 21
 pycccl.neutrinos (module), 22
 pycccl.power (module), 23
 pycccl.pyutils (module), 24

R

read_yaml() (pycccl.core.Cosmology class method), 19
 rho_x() (in module pycccl.background), 13

S

scale_factor_of_chi() (in module pycccl.background), 14

`sigma8()` (in module `pyccl.power`), 23
`sigmaM()` (in module `pyccl.massfunction`), 22
`sigmaR()` (in module `pyccl.power`), 23
`sigmaV()` (in module `pyccl.power`), 23
`status()` (`pyccl.core.Cosmology` method), 19

T

`Tracer` (class in `pyccl.cls`), 14

W

`WeakLensingTracer` (class in `pyccl.cls`), 15
`write_yaml()` (`pyccl.core.Cosmology` method), 19