
CAMB Python Documentation

Release 1.1.3

Antony Lewis

May 15, 2020

Contents

1 Basic functions	3
2 Input parameter model	7
3 Calculation results	17
4 Symbolic manipulation	31
5 BBN models	35
6 Dark Energy models	39
7 Initial power spectra	41
8 Non-linear models	45
9 Reionization models	47
10 Recombination models	49
11 Source windows functions	51
12 Correlation functions	53
13 Post-Born lensing	59
14 Lensing emission angle	61
15 Matter power spectrum and matter transfer function variables	63
16 Fortran compilers	65
17 Updating and modified Fortran code	67
18 Maths utils	69
Python Module Index	71
Index	73

CAMB (Code for Anisotropies in the Microwave Background), a cosmology code for calculating CMB, lensing, galaxy count, dark-age 21cm power spectra, matter power spectra and transfer functions. There are also general utility function for cosmological calculations such as the background expansion, distances, etc. The main code is Python with numerical calculations implemented efficiently in Python-wrapped modern Fortran.

See the [CAMB python example notebook](#) for an introductory set of examples of how to use the CAMB package. This is usually the fastest way to learn how to use it and quickly see some of the capabilities.

For a standard non-editable installation use:

```
pip install camb [--user]
```

The `--user` is optional and only required if you don't have write permission to your main python installation. If you want to work on the code from [GitHub](#), you can also just install in place without copying anything using:

```
pip install -e /path/to/CAMB [--user]
```

You will need ifort or gfortran 6 or higher installed (and on your path) to compile; see [Fortran compilers](#) for compiler installation details if needed. A compiled library for Windows is also provided, and is used if no gfortran installation is found on Windows machines. If you have gfortran installed, "python setup.py make" will build the Fortran library on all systems (including Windows without directly using a Makefile), and can be used to update a source installation after changes or pulling an updated version.

Anaconda users can also install from conda-forge using:

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

with no need for a Fortran compiler (unless you want to use custom sources/symbolic compilation features). Check that conda installs the latest version, if not try installing in a new clean conda environment.

After installation the camb python module can be loaded from your scripts using "import camb". You can also run CAMB from the command line reading parameters from a .ini file, e.g.:

```
camb infiles/planck_2018.ini
```

You may need to check your python scripts directory is in your path for this to work. Alternatively from the source package root directory (after make but without installation) use:

```
python camb.py infiles/planck_2018.ini
```

Main high-level modules:

Python CAMB interface (<https://camb.info>)

`camb.get_results(params)`

Calculate results for specified parameters and return *CAMBdata* instance for getting results.

Parameters `params` – *model.CAMBparams* instance

Returns *CAMBdata* instance

`camb.get_background(params, no_thermo=False)`

Calculate background cosmology for specified parameters and return *CAMBdata*, ready to get derived parameters and use background functions like *angular_diameter_distance()*.

Parameters

- `params` – *model.CAMBparams* instance
- `no_thermo` – set True if thermal and ionization history not required.

Returns *CAMBdata* instance

`camb.get_transfer_functions(params, only_time_sources=False)`

Calculate transfer functions for specified parameters and return *CAMBdata* instance for getting results and subsequently calculating power spectra.

Parameters

- `params` – *model.CAMBparams* instance
- `only_time_sources` – does not calculate the CMB l,k transfer functions and does not apply any non-linear correction scaling. Results with `only_time_sources=True` can therefore be used with different initial power spectra to get consistent non-linear lensed spectra.

Returns *CAMBdata* instance

`camb.set_params(cp=None, verbose=False, **params)`

Set all CAMB parameters at once, including parameters which are part of the *CAMBparams* structure, as well as global parameters.

E.g.:

```
cp = camb.set_params(ns=1, H0=67, ombh2=0.022, omch2=0.1, w=-0.95, Alens=1.2,
↳lmax=2000,
                    WantTransfer=True, dark_energy_model='DarkEnergyPPF')
```

This is equivalent to:

```
cp = model.CAMBparams()
cp.DarkEnergy = DarkEnergyPPF()
cp.DarkEnergy.set_params(w=-0.95)
cp.set_cosmology(H0=67, omch2=0.1, ombh2=0.022, Alens=1.2)
cp.set_for_lmax(lmax=2000)
cp.InitPower.set_params(ns=1)
cp.WantTransfer = True
```

The wrapped functions are (in this order):

- `model.CAMBparams.set_accuracy()`
- `model.CAMBparams.set_classes()`
- `dark_energy.DarkEnergyEqnOfState.set_params()` (or equivalent if a different dark energy model class used)
- `model.CAMBparams.set_cosmology()`
- `model.CAMBparams.set_matter_power()`
- `model.CAMBparams.set_for_lmax()`
- `initialpower.InitialPowerLaw.set_params()` (or equivalent if a different initial power model class used)
- `nonlinear.Halofit.set_params()`

Parameters

- **params** – the values of the parameters
- **cp** – use this CAMBparams instead of creating a new one
- **verbose** – print out the equivalent set of commands

Returns `model.CAMBparams` instance

```
camb.set_params_cosmomic(p, num_massive_neutrinos=1, neutrino_hierarchy='degenerate',
                        halofit_version='mead', dark_energy_model='ppf', lmax=2500,
                        lens_potential_accuracy=1, inpars=None)
```

get CAMBParams for dictionary of cosmomic-named parameters assuming Planck 2018 defaults

Parameters

- **p** – dictionary of cosmomic parameters (e.g. from `getdist.types.BestFit`'s `getParamDict()` function)
- **num_massive_neutrinos** – usually 1 if fixed $m_{\nu}=0.06$ eV, three if m_{ν} varying
- **neutrino_hierarchy** – hierarchy
- **halofit_version** – name of the specific Halofit model to use for non-linear modelling
- **dark_energy_model** – ppf or fluid dark energy model
- **lmax** – lmax for accuracy settings
- **lens_potential_accuracy** – lensing accuracy parameter

- **inpars** – optional input CAMBParams to set

Returns

`camb.read_ini(ini_filename, no_validate=False)`

Get a `model.CAMBparams` instance using parameter specified in a .ini parameter file.

Parameters

- **ini_filename** – path of the .ini file to read
- **no_validate** – do not pre-validate the ini file (faster, but may crash kernel if error)

Returns `model.CAMBparams` instance

`camb.get_matter_power_interpolator(params, zmin=0, zmax=10, nz_step=100, zs=None, kmax=10, nonlinear=True, var1=None, var2=None, hubble_units=True, k_hunit=True, return_z_k=False, k_per_logint=None, log_interp=True, extrap_kmax=None)`

Return a 2D spline interpolation object to evaluate matter power spectrum as function of z and k/h, e.g.

```
from camb import get_matter_power_interpolator
PK = get_matter_power_interpolator(params);
print('Power spectrum at z=0.5, k/h=0.1/Mpc is %s (Mpc/h)^3'%(PK.P(0.5, 0.1)))
```

For a description of outputs for different var1, var2 see *Matter power spectrum and matter transfer function variables*. If you already have a `CAMBdata` result object, you can instead use `get_matter_power_interpolator()`.

Parameters

- **params** – `model.CAMBparams` instance
- **zmin** – minimum z (use 0 or smaller than you want for good interpolation)
- **zmax** – maximum z (use larger than you want for good interpolation)
- **nz_step** – number of steps to sample in z (default max allowed is 100)
- **zs** – instead of zmin,zmax, nz_step, can specific explicit array of z values to spline from
- **kmax** – maximum k
- **nonlinear** – include non-linear correction from halo model
- **var1** – variable i (index, or name of variable; default delta_tot)
- **var2** – variable j (index, or name of variable; default delta_tot)
- **hubble_units** – if true, output power spectrum in $(\text{Mpc}/h)^3$ units, otherwise Mpc^3
- **k_hunit** – if true, matter power is a function of k/h, if false, just k (both Mpc^{-1} units)
- **return_z_k** – if true, return interpolator, z, k where z, k are the grid used
- **k_per_logint** – specific uniform sampling over log k (if not set, uses optimized irregular sampling)
- **log_interp** – if true, interpolate log of power spectrum (unless any values are negative in which case ignored)
- **extrap_kmax** – if set, use power law extrapolation beyond kmax to extrap_kmax (useful for tails of integrals)

Returns An object PK based on `RectBivariateSpline`, that can be called with `PK.P(z,kh)` or `PK(z,log(kh))` to get log matter power values. If `return_z_k=True`, instead return interpolator, z, k where z, k are the grid used.

`camb.get_age(params)`

Get age of universe for given set of parameters

Parameters `params` – `model.CAMBparams` instance

Returns age of universe in gigayears

`camb.get_zre_from_tau(params, tau)`

Get reionization redshift given optical depth tau

Parameters

- `params` – `model.CAMBparams` instance
- `tau` – optical depth

Returns reionization redshift (or negative number if error)

`camb.set_feedback_level(level=1)`

Set the feedback level for internal CAMB calls

Parameters `level` – zero for nothing, >1 for more

`camb.run_ini(ini_filename, no_validate=False)`

Run the command line camb from a .ini file (producing text files as with the command line program). This does the same as the command line program, except global config parameters are not read and set (which does not change results in almost all cases).

Parameters

- `ini_filename` – .ini file to use
- `no_validate` – do not pre-validate the ini file (faster, but may crash kernel if error)

`camb.get_valid_numerical_params(transfer_only=False, **class_names)`

Get numerical parameter names that are valid input to `set_params()`

Parameters

- `transfer_only` – if True, exclude parameters that affect only initial power spectrum or non-linear model
- `class_names` – class name parameters that will be used by `model.CAMBparams.set_classes()`

Returns set of valid input parameter names for `set_params()`

Input parameter model

class `camb.model.CAMBparams` (***kwargs*)

Object storing the parameters for a CAMB calculation, including cosmological parameters and settings for what to calculate. When a new object is instantiated, default parameters are set automatically.

To add a new parameter, add it to the CAMBparams type in `model.f90`, then edit the `_fields_` list in the CAMBparams class in `model.py` to add the new parameter in the corresponding location of the member list. After rebuilding the python version you can then access the parameter by using `params.new_parameter_name` where `params` is a CAMBparams instance. You could also modify the wrapper functions to set the field value less directly.

You can view the set of underlying parameters used by the Fortran code by printing the CAMBparams instance. In python, to set cosmology parameters it is usually best to use `set_cosmology()` and equivalent methods for most other parameters. Alternatively the convenience function `camb.set_params()` can construct a complete instance from a dictionary of relevant parameters.

Variables

- **WantCls** – (*boolean*) Calculate C_L
- **WantTransfer** – (*boolean*) Calculate matter transfer functions and matter power spectrum
- **WantScalars** – (*boolean*) Calculates scalar modes
- **WantTensors** – (*boolean*) Calculate tensor modes
- **WantVectors** – (*boolean*) Calculate vector modes
- **WantDerivedParameters** – (*boolean*) Calculate derived parameters
- **Want_cl_2D_array** – (*boolean*) For the C_L, include NxN matrix of all possible cross-spectra between sources
- **Want_CMB** – (*boolean*) Calculate the temperature and polarization power spectra
- **Want_CMB_lensing** – (*boolean*) Calculate the lensing potential power spectrum
- **DoLensing** – (*boolean*) Include CMB lensing

- **NonLinear** – (integer/string, one of: NonLinear_none, NonLinear_pk, NonLinear_lens, NonLinear_both)
- **Transfer** – *camb.model.TransferParams*
- **want_zstar** – (boolean)
- **want_zdrag** – (boolean)
- **min_l** – (integer) l_{\min} for the scalar C_L (1 or 2, $L=1$ dipoles are Newtonian Gauge)
- **max_l** – (integer) l_{\max} for the scalar C_L
- **max_l_tensor** – (integer) l_{\max} for the tensor C_L
- **max_eta_k** – (float64) Maximum $k \cdot \eta_0$ for scalar C_L , where η_0 is the conformal time today
- **max_eta_k_tensor** – (float64) Maximum $k \cdot \eta_0$ for tensor C_L , where η_0 is the conformal time today
- **ombh2** – (float64) $\Omega_{\text{baryon}} h^2$
- **omch2** – (float64) $\Omega_{\text{cdm}} h^2$
- **omk** – (float64) Ω_K
- **omnuh2** – (float64) $\Omega_{\text{massive_neutrino}} h^2$
- **H0** – (float64) Hubble parameter in km/s/Mpc units
- **TCMB** – (float64) CMB temperature today in Kelvin
- **YHe** – (float64) Helium mass fraction
- **num_nu_massless** – (float64) Effective number of massless neutrinos
- **num_nu_massive** – (integer) Total physical (integer) number of massive neutrino species
- **nu_mass_eigenstates** – (integer) Number of non-degenerate mass eigenstates
- **share_delta_neff** – (boolean) Share the non-integer part of `num_nu_massless` between the eigenstates
- **nu_mass_degeneracies** – (float64 array) Degeneracy of each distinct eigenstate
- **nu_mass_fractions** – (float64 array) Mass fraction in each distinct eigenstate
- **nu_mass_numbers** – (integer array) Number of physical neutrinos per distinct eigenstate
- **InitPower** – *camb.initialpower.InitialPower*
- **Recomb** – *camb.recombination.RecombinationModel*
- **Reion** – *camb.reionization.ReionizationModel*
- **DarkEnergy** – *camb.dark_energy.DarkEnergyModel*
- **NonLinearModel** – *camb.nonlinear.NonLinearModel*
- **Accuracy** – *camb.model.AccuracyParams*
- **SourceTerms** – *camb.model.SourceTermParams*
- **z_outputs** – (float64 array) redshifts to always calculate BAO output parameters
- **scalar_initial_condition** – (integer/string, one of: initial_vector, initial_adiabatic, initial_iso_CDM, initial_iso_baryon, initial_iso_neutrino, initial_iso_neutrino_vel)

- **InitialConditionVector** – (*float64 array*) if `scalar_initial_condition` is `initial_vector`, the vector of initial condition amplitudes
- **OutputNormalization** – (*integer*) If non-zero, multipole to normalize the C_L at
- **Alens** – (*float64*) non-physical scaling amplitude for the CMB lensing spectrum power
- **MassiveNuMethod** – (*integer/string*, one of: `Nu_int`, `Nu_trunc`, `Nu_approx`, `Nu_best`)
- **DoLateRadTruncation** – (*boolean*) If true, use smooth approx to radiation perturbations after decoupling on small scales, saving evolution of irrelevant oscillatory multipole equations
- **Evolve_baryon_cs** – (*boolean*) Evolve a separate equation for the baryon sound speed rather than using background approximation
- **Evolve_delta_xe** – (*boolean*) Evolve ionization fraction perturbations
- **Evolve_delta_Ts** – (*boolean*) Evolve the spin temperature perturbation (for 21cm)
- **Do21cm** – (*boolean*) 21cm is not yet implemented via the python wrapper
- **transfer_21cm_cl** – (*boolean*) Get 21cm C_L at a given fixed redshift
- **Log_lvalues** – (*boolean*) Use log spacing for sampling in L
- **use_cl_spline_template** – (*boolean*) When interpolating use a fiducial spectrum shape to define ratio to spline
- **SourceWindows** – array of `camb.sources.SourceWindow`
- **CustomSources** – `camb.model.CustomSources`

N_eff

Returns Effective number of degrees of freedom in relativistic species at early times.

copy()

Make independent copy of this object.

Returns deep copy of self

diff(params)

Print differences between this set of parameters and params

Parameters `params` – another CAMBparams instance

get_DH(ombh2=None, delta_neff=None)

Get deuterium ration D/H by interpolation using the `bbn.BBNPredictor` instance passed to `set_cosmology()` (or the default one, if `Y_He` has not been set).

Parameters

- **ombh2** – $\Omega_b h^2$ (default: value passed to `set_cosmology()`)
- **delta_neff** – additional N_{eff} relative to standard value (of 3.046) (default: from values passed to `set_cosmology()`)

Returns BBN helium nucleon fraction D/H

get_Y_p(ombh2=None, delta_neff=None)

Get BBN helium nucleon fraction (NOT the same as the mass fraction `Y_He`) by interpolation using the `bbn.BBNPredictor` instance passed to `set_cosmology()` (or the default one, if `Y_He` has not been set).

Parameters

- **ombh2** – $\Omega_b h^2$ (default: value passed to `set_cosmology()`)
- **delta_neff** – additional N_{eff} relative to standard value (of 3.046) (default: from values passed to `set_cosmology()`)

Returns Y_p^{BBN} helium nucleon fraction predicted by BBN.

replace (*instance*)

Replace the content of this class with another instance, doing a deep copy (in Fortran)

Parameters **instance** – instance of the same class to replace this instance with

scalar_power (*k*)

Get the primordial scalar curvature power spectrum at *k*

Parameters **k** – wavenumber *k* (in Mpc^{-1} units)

Returns power spectrum at *k*

set_H0_for_theta (*theta*, *cosmomc_approx=False*, *theta_H0_range=(10, 100)*, *est_H0=67.0*, *iteration_threshold=8*)

Set H0 to give a specified value of the acoustic angular scale parameter theta.

Parameters

- **theta** – value of r_s/D_M at redshift z_*
- **cosmomc_approx** – if true, use approximate fitting formula for z_* , if false do full numerical calculation
- **theta_H0_range** – min, max interval to search for H0 (in km/s/Mpc)
- **est_H0** – an initial guess for H0 in km/s/Mpc, used in the case `cosmomc_approx=False`.
- **iteration_threshold** – difference in H0 from `est_H0` for which to iterate, used for `cosmomc_approx=False`

set_accuracy (*AccuracyBoost=1.0*, *lSampleBoost=1.0*, *lAccuracyBoost=1.0*, *DoLateRadTruncation=True*)

Set parameters determining overall calculation accuracy (large values may give big slow down). For finer control you can set individual accuracy parameters by changing `CAMBParams.Accuracy(model.AccuracyParams)`.

Parameters

- **AccuracyBoost** – increase AccuracyBoost to decrease integration step size, increase density of k sampling, etc.
- **lSampleBoost** – increase lSampleBoost to increase density of L sampling for CMB
- **lAccuracyBoost** – increase lAccuracyBoost to increase the maximum L included in the Boltzmann hierarchies
- **DoLateRadTruncation** – If True, use approximation to radiation perturbation evolution at late times

Returns self

set_classes (*dark_energy_model=None*, *initial_power_model=None*, *non_linear_model=None*, *recombination_model=None*)

Change the classes used to implement parts of the model.

Parameters

- **dark_energy_model** – ‘fluid’, ‘ppf’, or name of a DarkEnergyModel class
- **initial_power_model** – name of an InitialPower class

- **non_linear_model** – name of a NonLinearModel class
- **recombination_model** – name of recombination_model class

set_cosmology (*H0: Optional[float] = None, ombh2=0.022, omch2=0.12, omk=0.0, cosmomc_theta: Optional[float] = None, thetastar: Optional[float] = None, neutrino_hierarchy: Union[str, int] = 'degenerate', num_massive_neutrinos=1, mnu=0.06, nnu=3.046, YHe: Optional[float] = None, meffsterile=0.0, standard_neutrino_neff=3.046, TCMB=2.7255, tau: Optional[float] = None, zrei: Optional[float] = None, deltazrei: Optional[float] = None, Alens=1.0, bbn_predictor: Union[None, str, camb.bbn.BBNPredictor] = None, theta_H0_range=(10, 100)*)

Sets cosmological parameters in terms of physical densities and parameters (e.g. as used in Planck analyses). Default settings give a single distinct neutrino mass eigenstate, by default one neutrino with $m_{\nu} = 0.06\text{eV}$. Set the `neutrino_hierarchy` parameter to `normal` or `inverted` to use a two-eigenstate model that is a good approximation to the known mass splittings seen in oscillation measurements. For more fine-grained control can set the neutrino parameters directly rather than using this function.

Instead of setting the Hubble parameter directly, you can instead set the acoustic scale parameter (`cosmomc_theta`, which is based on a fitting formula for simple models, or `thetastar`, which is numerically calculated more generally). Note that you must have already set the dark energy model, you can't use `set_cosmology` with `theta` and then change the background evolution (which would change `theta` at the calculated `H0` value). Likewise the dark energy model cannot depend explicitly on `H0`.

Parameters

- **H0** – Hubble parameter today in km/s/Mpc. Can leave unset and instead set `thetastar` or `cosmomc_theta` (which solves for the required `H0`).
- **ombh2** – physical density in baryons
- **omch2** – physical density in cold dark matter
- **omk** – Omega_K curvature parameter
- **cosmomc_theta** – The approximate CosmoMC theta parameter θ_{MC} . The angular diameter distance is calculated numerically, but the redshift z_* is calculated using an approximate (quite accurate but non-general) fitting formula. Leave unset to use `H0` or `thetastar`.
- **thetastar** – The angular acoustic scale parameter $\theta_* = r_s(z_*)/D_M(z_*)$, defined as the ratio of the photon-baryon sound horizon r_s to the angular diameter distance D_M , where both quantities are evaluated at z_* , the redshift at which the optical depth (excluding reionization) is unity. Leave unset to use `H0` or `cosmomc_theta`.
- **neutrino_hierarchy** – ‘degenerate’, ‘normal’, or ‘inverted’ (1 or 2 eigenstate approximation)
- **num_massive_neutrinos** – number of massive neutrinos
- **mnu** – sum of neutrino masses (in eV). `Omega_nu` is calculated approximately from this assuming neutrinos non-relativistic today; i.e. here is defined as a direct proxy for `Omega_nu`. Internally the actual physical mass is calculated from the `Omega_nu` accounting for small mass-dependent velocity corrections but neglecting spectral distortions to the neutrino distribution. Set the neutrino field values directly if you need finer control or more complex neutrino models.
- **nnu** – `N_eff`, effective relativistic degrees of freedom
- **YHe** – Helium mass fraction. If `None`, set from BBN consistency.
- **meffsterile** – effective mass of sterile neutrinos

- **standard_neutrino_neff** – default value for N_{eff} in standard cosmology (non-integer to allow for partial heating of neutrinos at electron-positron annihilation and QED effects)
- **TCMB** – CMB temperature (in Kelvin)
- **tau** – optical depth; if None and **zrei** is None, current Reion settings are not changed
- **zrei** – reionization mid-point optical depth (set **tau**=None to use this)
- **deltazrei** – redshift width of reionization; if None, uses default
- **Alens** – (non-physical) scaling of the lensing potential compared to prediction
- **bbn_predictor** – `bbn.BBNPredictor` instance used to get YHe from BBN consistency if YHe is None, or name of a BBN predictor class, or file name of an interpolation table
- **theta_H0_range** – if **thetastar** or **cosmomc_theta** is specified, the min, max interval of H_0 values to map to; if H_0 is outside this range it will raise an exception.

set_custom_scalar_sources (*custom_sources*, *source_names=None*, *source_ell_scales=None*, *frame='CDM'*, *code_path=None*)

Set custom sources for angular power spectrum using `camb.symbolic` sympy expressions.

Parameters

- **custom_sources** – list of sympy expressions for the angular power spectrum sources
- **source_names** – optional list of string names for the sources
- **source_ell_scales** – list or dictionary of scalings for each source name, where for integer entry n , the source for multipole ℓ is scaled by $\sqrt{(\ell + n)!/(\ell - n)!}$, i.e. $n = 2$ for a new polarization-like source.
- **frame** – if the source is not gauge invariant, frame in which to interpret result
- **code_path** – optional path for output of source code for CAMB f90 source function

set_dark_energy (*w=-1.0*, *cs2=1.0*, *wa=0*, *dark_energy_model='fluid'*)

Set dark energy parameters (use `set_dark_energy_w_a` to set $w(a)$ from numerical table instead) To use a custom dark energy model, assign the class instance to the DarkEnergy field instead.

Parameters

- **w** – $w \equiv p_{\text{de}}/\rho_{\text{de}}$, assumed constant
- **wa** – evolution of w (for `dark_energy_model=ppf`)
- **cs2** – rest-frame sound speed squared of dark energy fluid
- **dark_energy_model** – model to use ('fluid' or 'ppf'), default is 'fluid'

Returns

self

set_dark_energy_w_a (*a*, *w*, *dark_energy_model='fluid'*)

Set the dark energy equation of state from tabulated values (which are cubic spline interpolated).

Parameters

- **a** – array of sampled $a = 1/(1+z)$ values
- **w** – array of $w(a)$
- **dark_energy_model** – model to use ('fluid' or 'ppf'), default is 'fluid'

Returns

self

set_for_lmax (*lmax*, *max_eta_k=None*, *lens_potential_accuracy=0*, *lens_margin=150*,
k_eta_fac=2.5, *lens_k_eta_reference=18000.0*)

Set parameters to get CMB power spectra accurate to specific a *l_max*. Note this does not fix the actual output L range, spectra may be calculated above *l_max* (but may not be accurate there). To fix the *l_max* for output arrays use the optional input argument to `results.CAMBdata.get_cmb_power_spectra()` etc.

Parameters

- **lmax** – ℓ_{\max} you want
- **max_eta_k** – maximum value of $k\eta_0 \approx k\chi_*$ to use, which indirectly sets *k_max*. If None, sensible value set automatically.
- **lens_potential_accuracy** – Set to 1 or higher if you want to get the lensing potential accurate (1 is only Planck-level accuracy)
- **lens_margin** – the $\Delta\ell_{\max}$ to use to ensure lensed C_ℓ are correct at ℓ_{\max}
- **k_eta_fac** – *k_eta_fac* default factor for setting *max_eta_k* = *k_eta_fac***lmax* if *max_eta_k*=None
- **lens_k_eta_reference** – value of *max_eta_k* to use when *lens_potential_accuracy*>0; use *k_eta_max* = *lens_k_eta_reference***lens_potential_accuracy*

Returns self

set_initial_power (*initial_power_params*)

Set the InitialPower primordial power spectrum parameters

Parameters **initial_power_params** – `initialpower.InitialPowerLaw` or `initialpower.SplinedInitialPower` instance

Returns self

set_initial_power_function (*P_scalar*, *P_tensor=None*, *kmin=1e-06*, *kmax=100.0*,
N_min=200, *rtol=5e-05*, *effective_ns_for_nonlinear=None*,
args=())

Set the initial power spectrum from a function *P_scalar*(*k*, **args*), and optionally also the tensor spectrum. The function is called to make a pre-computed array which is then interpolated inside CAMB. The sampling in *k* is set automatically so that the spline is accurate, but you may also need to increase other accuracy parameters.

Parameters

- **P_scalar** – function returning normalized initial scalar curvature power as function of *k* (in Mpc^{-1})
- **P_tensor** – optional function returning normalized initial tensor power spectrum
- **kmin** – minimum wavenumber to compute
- **kmax** – maximum wavenumber to compute
- **N_min** – minimum number of spline points for the pre-computation
- **rtol** – relative tolerance for deciding how many points are enough
- **effective_ns_for_nonlinear** – an effective *n_s* for use with approximate non-linear corrections
- **args** – optional list of arguments passed to *P_scalar* (and *P_tensor*)

Returns self

set_initial_power_table (*k*, *pk=None*, *pk_tensor=None*, *effective_ns_for_nonlinear=None*)

Set a general initial power spectrum from tabulated values. It's up to you to ensure the sampling of the *k* values is high enough that it can be interpolated accurately.

Parameters

- **k** – array of *k* values (Mpc^{-1})
- **pk** – array of primordial curvature perturbation power spectrum values $P(k_i)$
- **pk_tensor** – array of tensor spectrum values
- **effective_ns_for_nonlinear** – an effective *n_s* for use with approximate non-linear corrections

set_matter_power (*redshifts=(0.0,)*, *kmax=1.2*, *k_per_logint=None*, *nonlinear=None*, *accurate_massive_neutrino_transfers=False*, *silent=False*)

Set parameters for calculating matter power spectra and transfer functions.

Parameters

- **redshifts** – array of redshifts to calculate
- **kmax** – maximum *k* to calculate (where *k* is just *k*, not *k/h*)
- **k_per_logint** – minimum number of *k* steps per log *k*. Set to zero to use default optimized spacing.
- **nonlinear** – if *None*, uses existing setting, otherwise boolean for whether to use non-linear matter power.
- **accurate_massive_neutrino_transfers** – if you want the massive neutrino transfers accurately
- **silent** – if *True*, don't give warnings about sort order

Returns *self*

set_nonlinear_lensing (*nonlinear*)

Settings for whether or not to use non-linear corrections for the CMB lensing potential. Note that *set_for_lmax* also sets lensing to be non-linear if *lens_potential_accuracy*>0

Parameters **nonlinear** – true to use non-linear corrections

tensor_power (*k*)

Get the primordial tensor curvature power spectrum at *k*

Parameters **k** – wavenumber *k* (in Mpc^{-1} units)

Returns tensor power spectrum at *k*

validate ()

Do some quick tests for sanity

Returns *True* if OK

class `camb.model.AccuracyParams`

Structure with parameters governing numerical accuracy. *AccuracyBoost* will also scale almost all the other parameters except for *lSampleBoost* (which is specific to the output interpolation) and *lAccuracyBoost* (which is specific to the multipole hierarchy evolution), e.g setting *AccuracyBoost*=2, *IntTolBoost*=1.5, means that internally the *k* sampling for integration will be boosted by *AccuracyBoost***IntTolBoost* = 3.

Variables

- **AccuracyBoost** – (*float64*) general accuracy setting effecting everything related to step sizes etc. (including separate settings below except the next two)

- **lSampleBoost** – (*float64*) accuracy for sampling in ell for interpolation for the C_l (if ≥ 50 , all ell are calculated)
- **lAccuracyBoost** – (*float64*) Boosts number of multipoles integrated in Boltzman heirarchy
- **AccuratePolarization** – (*boolean*) Do you care about the accuracy of the polarization Cls?
- **AccurateBB** – (*boolean*) Do you care about BB accuracy (e.g. in lensing)
- **AccurateReionization** – (*boolean*) Do you care about percent level accuracy on EE signal from reionization?
- **TimeStepBoost** – (*float64*) Sampling timesteps
- **BackgroundTimeStepBoost** – (*float64*) Number of time steps for background thermal history and source window interpolation
- **IntTolBoost** – (*float64*) Tolerances for integrating differential equations
- **SourcekAccuracyBoost** – (*float64*) Accuracy of k sampling for source time integration
- **IntkAccuracyBoost** – (*float64*) Accuracy of k sampling for integration
- **TransferkBoost** – (*float64*) Accuracy of k sampling for transfer functions
- **NonFlatIntAccuracyBoost** – (*float64*) Accuracy of non-flat time integration
- **BessIntBoost** – (*float64*) Accuracy of bessel integration truncation
- **LensingBoost** – (*float64*) Accuracy of the lensing of CMB power spectra
- **NonlinSourceBoost** – (*float64*) Accuracy of steps and kmax used for the non-linear correction
- **BesselBoost** – (*float64*) Accuracy of bessel pre-computation sampling
- **LimberBoost** – (*float64*) Accuracy of Limber approximation use
- **SourceLimberBoost** – (*float64*) Scales when to switch to Limber for source windows
- **KmaxBoost** – (*float64*) Boost max k for source window functions
- **neutrino_q_boost** – (*float64*) Number of momenta integrated for neutrino perturbations

class camb.model.TransferParams

Object storing parameters for the matter power spectrum calculation.

Variables

- **high_precision** – (*boolean*) True for more accuracy
- **accurate_massive_neutrinos** – (*boolean*) True if you want neutrino transfer functions accurate (false by default)
- **kmax** – (*float64*) k_max to output (no h in units)
- **k_per_logint** – (*integer*) number of points per log k interval. If zero, set an irregular optimized spacing
- **PK_num_redshifts** – (*integer*) number of redshifts to calculate
- **PK_redshifts** – (*float64 array*) redshifts to output for the matter transfer and power

class `camb.model.SourceTermParams`

Structure with parameters determining how galaxy/lensing/21cm power spectra and transfer functions are calculated.

Variables

- **limber_windows** – (*boolean*) Use Limber approximation where appropriate. CMB lensing uses Limber even if `limber_window` is false, but method is changed to be consistent with other sources if `limber_windows` is true
- **limber_phi_lmin** – (*integer*) When `limber_windows=True`, the minimum L to use Limber approximation for the lensing potential and other sources (which may use higher but not lower)
- **counts_density** – (*boolean*) Include the density perturbation source
- **counts_redshift** – (*boolean*) Include redshift distortions
- **counts_lensing** – (*boolean*) Include magnification bias for number counts
- **counts_velocity** – (*boolean*) Non-redshift distortion velocity terms
- **counts_radial** – (*boolean*) Radial displacement velocity term; does not include time delay; subset of `counts_velocity`, just $1 / (\chi * H)$ term
- **counts_timedelay** – (*boolean*) Include time delay terms $* 1 / (H * \chi)$
- **counts_ISW** – (*boolean*) Include tiny ISW terms
- **counts_potential** – (*boolean*) Include tiny terms in potentials at source
- **counts_evolve** – (*boolean*) Account for source evolution
- **line_phot_dipole** – (*boolean*) Dipole sources for 21cm
- **line_phot_quadrupole** – (*boolean*) Quadrupole sources for 21cm
- **line_basic** – (*boolean*) Include main 21cm monopole density/spin temperature sources
- **line_distortions** – (*boolean*) Redshift distortions for 21cm
- **line_extra** – (*boolean*) Include other sources
- **line_reionization** – (*boolean*) Replace the E modes with 21cm polarization
- **use_21cm_mK** – (*boolean*) Use mK units for 21cm

class `camb.model.CustomSources`

Structure containing symbolic-compiled custom CMB angular power spectrum source functions. Don't change this directly, instead call `model.CAMBparams.set_custom_scalar_sources()`.

Variables

- **num_custom_sources** – (*integer*) number of sources set
- **c_source_func** – (*pointer*) Don't directly change this
- **custom_source_ell_scales** – (*integer array*) scaling in L for outputs

Calculation results

class `camb.results.CAMBdata`

An object for storing calculational data, parameters and transfer functions. Results for a set of parameters (given in a `CAMBparams` instance) are returned by the `camb.get_background()`, `camb.get_transfer_functions()` or `camb.get_results()` functions. Exactly which quantities are already calculated depends on which of these functions you use and the input parameters.

To quickly make a fully calculated `CAMBdata` instance for a set of parameters you can call `camb.get_results()`.

Variables

- **Params** – `camb.model.CAMBparams`
- **ThermoDerivedParams** – (*float64 array*) array of derived parameters, see `get_derived_params()` to get as a dictionary
- **flat** – (*boolean*) flat universe
- **closed** – (*boolean*) closed universe
- **grhocrit** – (*float64*) $\kappa a^2 \rho_c(0)/c^2$ with units of $\text{Mpc}^{**(-2)}$
- **grhog** – (*float64*) $\kappa/c^2 * \sigma_B/c^3 T_{\text{CMB}}^4$
- **grhor** – (*float64*) $7/8 * (4/11)^{(4/3)} * \text{grhog}$ (per massless neutrino species)
- **grhob** – (*float64*) baryon contribution
- **grhoc** – (*float64*) CDM contribution
- **grhov** – (*float64*) Dark energy contribution
- **grhornomass** – (*float64*) $\text{grhor} * \text{number of massless neutrino species}$
- **grhok** – (*float64*) curvature contribution to critical density
- **taurst** – (*float64*) time at start of recombination
- **dtarec** – (*float64*) time step in recombination
- **taurend** – (*float64*) time at end of recombination

- **tau_maxvis** – (*float64*) time at peak visibility
- **adotrad** – (*float64*) da/d tau in early radiation-dominated era
- **omega_de** – (*float64*) Omega for dark energy today
- **curv** – (*float64*) curvature K
- **curvature_radius** – (*float64*) $1/\sqrt{|K|}$
- **Ksign** – (*float64*) Ksign = 1,0 or -1
- **tau0** – (*float64*) conformal time today
- **chi0** – (*float64*) comoving angular diameter distance of big bang; $\text{rofChi}(\text{tau0}/\text{curvature_radius})$
- **scale** – (*float64*) relative to flat. e.g. for scaling L sampling
- **akthom** – (*float64*) $\sigma_T * (\text{number density of protons now})$
- **fHe** – (*float64*) $n_{\text{He_tot}} / n_{\text{H_tot}}$
- **Nnow** – (*float64*) number density today
- **z_eq** – (*float64*) matter-radiation equality redshift assuming all neutrinos relativistic
- **grhormass** – (*float64 array*)
- **nu_masses** – (*float64 array*)
- **num_transfer_redshifts** – (*integer*) Number of calculated redshift outputs for the matter transfer (including those for CMB lensing)
- **transfer_redshifts** – (*float64 array*) Calculated output redshifts
- **PK_redshifts_index** – (*integer array*) Indices of the requested PK_redshifts
- **OnlyTransfers** – (*boolean*) Only calculating transfer functions, not power spectra
- **HasScalarTimeSources** – (*boolean*) calculate and save time source functions, not power spectra

angular_diameter_distance (*z*)

Get (non-comoving) angular diameter distance to redshift *z*.

Must have called `calc_background()`, `calc_background_no_thermo()` or calculated transfer functions or power spectra.

Parameters *z* – redshift or array of redshifts

Returns angular diameter distances, matching rank of *z*

angular_diameter_distance2 (*z1, z2*)

Get angular diameter distance between two redshifts $\frac{r}{1+z_2} \sin_K \left(\frac{\chi(z_2) - \chi(z_1)}{r} \right)$ where *r* is curvature radius and χ is the comoving radial distance.

Must have called `calc_background()`, `calc_background_no_thermo()` or calculated transfer functions or power spectra.

Parameters

- **z1** – redshift 1
- **z2** – redshift 2

Returns result

calc_background (*params*)

Calculate the background evolution and thermal history. e.g. call this if you want to get derived parameters and call background functions

Parameters *params* – *CAMBparams* instance to use

calc_background_no_thermo (*params*, *do_reion=False*)

Calculate the background evolution without calculating thermal or ionization history. e.g. call this if you want to just use *angular_diameter_distance()* and similar background functions

Parameters

- **params** – *CAMBparams* instance to use
- **do_reion** – whether to initialize the reionization model

calc_power_spectra (*params=None*)

Calculates transfer functions and power spectra.

Parameters *params* – optional *CAMBparams* instance with parameters to use

calc_transfers (*params*, *only_transfers=True*, *only_time_sources=False*)

Calculate the transfer functions (for CMB and matter power, as determined by *params.WantCls*, *params.WantTransfer*).

Parameters

- **params** – *CAMBparams* instance with parameters to use
- **only_transfers** – only calculate transfer functions, no power spectra
- **only_time_sources** – only calculate time transfer functions, no (p,l,k) transfer functions or non-linear scaling

Returns non-zero if error, zero if OK

comoving_radial_distance (*z*, *tol=0.0001*)

Get comoving radial distance from us to redshift *z* in Mpc. This is efficient for arrays.

Must have called *calc_background()*, *calc_background_no_thermo()* or calculated transfer functions or power spectra.

Parameters

- **z** – redshift
- **tol** – numerical tolerance parameter

Returns comoving radial distance (Mpc)

conformal_time (*z*, *presorted=None*, *tol=None*)

Conformal time from hot big bang to redshift *z* in Megaparsec.

Parameters

- **z** – redshift or array of redshifts
- **presorted** – if True, redshifts already sorted to be monotonically increasing, if False decreasing, or if None unsorted. If presorted is True or False no checks are done.
- **tol** – integration tolerance

Returns $\eta(z)/\text{Mpc}$

conformal_time_a1_a2 (*a1*, *a2*)

Get conformal time between two scale factors (=comoving radial distance travelled by light on light cone)

Parameters

- **a1** – scale factor 1
- **a2** – scale factor 2

Returns $\eta(a2) - \eta(a1) = \chi(a1) - \chi(a2)$ in Megaparsec

copy()

Make independent copy of this object.

Returns deep copy of self

cosmomc_theta()

Get θ_{MC} , an approximation of the ratio of the sound horizon to the angular diameter distance at recombination.

Returns θ_{MC}

get_BAO(redshifts, params)

Get BAO parameters at given redshifts, using parameters in params

Parameters

- **redshifts** – list of redshifts
- **params** – optional *CAMBparams* instance to use

Returns array of rs/DV, H, DA, F_AP for each redshift as 2D array

get_Omega(var, z=0)

Get density relative to critical density of variables var

Parameters

- **var** – one of 'K', 'cdm', 'baryon', 'photon', 'neutrino' (massless), 'nu' (massive neutrinos), 'de'
- **z** – redshift

Returns $\Omega_i(a)$

get_background_densities(a, vars=['tot', 'K', 'cdm', 'baryon', 'photon', 'neutrino', 'nu', 'de'], format='dict')

Get the individual densities as a function of scale factor. Returns $8\pi G a^4 \rho_i$ in Mpc units. Ω_i can be simply obtained by taking the ratio of the components to tot.

Parameters

- **a** – scale factor or array of scale factors
- **vars** – list of variables to output (default all)
- **format** – 'dict' or 'array', for either dict of 1D arrays indexed by name, or 2D array

Returns n_a x len(vars) 2D numpy array or dict of 1D arrays of $8\pi G a^4 \rho_i$ in Mpc units.

get_background_outputs()

Get BAO values for redshifts set in Params.z_outputs

Returns rs/DV, H, DA, F_AP for each requested redshift (as 2D array)

get_background_redshift_evolution(z, vars=['x_e', 'opacity', 'visibility', 'cs2b', 'T_b', 'dopacity', 'ddopacity', 'dvisibility', 'ddvisibility'], format='dict')

Get the evolution of background variables a function of redshift. For the moment a and H are rather perversely only available via *get_time_evolution()*

Parameters

- **z** – array of requested redshifts to output
- **vars** – list of variable names to output
- **format** – ‘dict’ or ‘array’, for either dict of 1D arrays indexed by name, or 2D array

Returns $n_{\text{eta}} \times \text{len}(\text{vars})$ 2D numpy array of outputs or dict of 1D arrays

get_background_time_evolution(*eta*, *vars*=[‘x_e’, ‘opacity’, ‘visibility’, ‘cs2b’, ‘T_b’, ‘dopacity’, ‘ddopacity’, ‘dvisibility’, ‘ddvisibility’], *format*=‘dict’)

Get the evolution of background variables a function of conformal time. For the moment *a* and *H* are rather perversely only available via `get_time_evolution()`

Parameters

- **eta** – array of requested conformal times to output
- **vars** – list of variable names to output
- **format** – ‘dict’ or ‘array’, for either dict of 1D arrays indexed by name, or 2D array

Returns $n_{\text{eta}} \times \text{len}(\text{vars})$ 2D numpy array of outputs or dict of 1D arrays

get_cmb_correlation_functions(*params*=None, *lmax*=None, *spectrum*=‘lensed_scalar’, *xvals*=None, *sampling_factor*=1)

Get the CMB correlation functions from the power spectra. By default evaluated at points $\cos(\theta) = xvals$ that are roots of Legendre polynomials, for accurate back integration with `correlations.corr2cl()`. If *xvals* is explicitly given, instead calculates correlations at provided $\cos(\theta)$ values.

Parameters

- **params** – optional `CAMBparams` instance with parameters to use. If None, must have previously set parameters and called `calc_power_spectra()` (e.g. if you got this instance using `camb.get_results()`),
- **lmax** – optional maximum L to use from the cls arrays
- **spectrum** – type of CMB power spectrum to get; default ‘lensed_scalar’, one of [‘total’, ‘unlensed_scalar’, ‘unlensed_total’, ‘lensed_scalar’, ‘tensor’]
- **xvals** – optional array of $\cos(\theta)$ values at which to calculate correlation function.
- **sampling_factor** – multiple of *lmax* for the Gauss-Legendre order if *xvals* not given (default 1)

Returns if *xvals* not given: *corrs*, *xvals*, *weights*; if *xvals* specified, just *corrs*. *corrs* is 2D array *corrs*[*i*, *ix*], where *ix*=0,1,2,3 are T, Q+U, Q-U and cross, and *i* indexes *xvals*

get_cmb_power_spectra(*params*=None, *lmax*=None, *spectra*=(‘total’, ‘unlensed_scalar’, ‘unlensed_total’, ‘lensed_scalar’, ‘tensor’, ‘lens_potential’), *CMB_unit*=None, *raw_cl*=False)

Get CMB power spectra, as requested by the ‘*spectra*’ argument. All power spectra are $\ell(\ell + 1)C_\ell/2\pi$ self-owned numpy arrays (0..lmax, 0..3), where 0..3 index are TT, EE, BB, TE, unless *raw_cl* is True in which case return just C_ℓ . For the *lens_potential* the power spectrum returned is that of the deflection.

Parameters

- **params** – optional `CAMBparams` instance with parameters to use. If None, must have previously set parameters and called `calc_power_spectra` (e.g. if you got this instance using `camb.get_results()`),
- **lmax** – maximum L

- **spectra** – list of names of spectra to get
- **CMB_unit** – scale results from dimensionless. Use ‘muK’ for μK^2 units for CMB C_ℓ and μK units for lensing cross.
- **raw_cl** – return C_ℓ rather than $\ell(\ell + 1)C_\ell/2\pi$

Returns dictionary of power spectrum arrays, indexed by names of requested spectra

get_cmb_transfer_data (*tp='scalar'*)

Get C_ℓ transfer functions

Returns *ClTransferData* instance holding output arrays (copies, not pointers)

get_cmb_unlensed_scalar_array_dict (*params=None, lmax=None, CMB_unit=None, raw_cl=False*)

Get all unlensed auto and cross power spectra, including any custom source functions set using *model.CAMBparams.set_custom_scalar_sources()*.

Parameters

- **params** – optional *CAMBparams* instance with parameters to use. If None, must have previously set parameters and called *calc_power_spectra()* (e.g. if you got this instance using *camb.get_results()*),
- **lmax** – maximum ℓ
- **CMB_unit** – scale results from dimensionless. Use ‘muK’ for μK^2 units for CMB C_ℓ and μK units for lensing cross.
- **raw_cl** – return C_ℓ rather than $\ell(\ell + 1)C_\ell/2\pi$

Returns dictionary of power spectrum arrays, index as TxT, TxE, PxW1, W1xW2, custom_name_1xT... etc. Note that P is the lensing deflection, lensing windows Wx give convergence.

get_dark_energy_rho_w (*a*)

Get dark energy density in units of the dark energy density today, and w=P/rho

Parameters **a** – scalar factor or array of scale factors

Returns rho, w arrays at redshifts 1/a-1 [or scalars if a is scalar]

get_derived_params ()

Returns dictionary of derived parameter values, indexed by name (‘kd’, ‘age’, etc..)

get_fsigma8 ()

Get $f\sigma_8$ growth values (must previously have calculated power spectra). For general models $f\sigma_8$ is defined as in the Planck 2015 parameter paper in terms of the velocity-density correlation: $\sigma_{vd}^2/\sigma_{dd}$ for $8h^{-1}$ Mpc spheres.

Returns array of f*sigma_8 values, in order of increasing time (decreasing redshift)

get_lens_potential_cls (*lmax=None, CMB_unit=None, raw_cl=False*)

Get lensing deflection angle potential power spectrum, and cross-correlation with T and E. Must have already calculated power spectra. Power spectra are $[L(L + 1)]^2 C_L^{\phi\phi}/2\pi$ and corresponding deflection cross-correlations.

Parameters

- **lmax** – lmax to output to
- **CMB_unit** – scale results from dimensionless. Use ‘muK’ for μK units for lensing cross.
- **raw_cl** – return lensing potential C_L rather than $[L(L + 1)]^2 C_L/2\pi$

Returns numpy array CL[0:lmax+1,0:3], where 0..2 indexes PP, PT, PE.

get_lensed_gradient_cls (*lmax=None, CMB_unit=None, raw_cl=False*)

Get lensed gradient scalar CMB power spectra in flat sky approximation ([arXiv:1101.2234](https://arxiv.org/abs/1101.2234)). Note that lmax used to calculate results may need to be substantially larger than the lmax output from this function (there is no extrapolation as in the main lensing routines). Lensed power spectra must be already calculated.

Parameters

- **lmax** – lmax to output to
- **CMB_unit** – scale results from dimensionless. Use ‘muK’ for μK^2 units for CMB C_ℓ
- **raw_cl** – return C_ℓ rather than $\ell(\ell + 1)C_\ell/2\pi$

Returns numpy array CL[0:lmax+1,0:8], where CL[:,i] are $T\nabla T$, $E\nabla E$, $B\nabla B$, PP_\perp , $T\nabla E$, TP_\perp , $(\nabla T)^2$, $\nabla T\nabla T$ where the first six are as defined in appendix C of [1101.2234](https://arxiv.org/abs/1101.2234).

get_lensed_scalar_cls (*lmax=None, CMB_unit=None, raw_cl=False*)

Get lensed scalar CMB power spectra. Must have already calculated power spectra.

Parameters

- **lmax** – lmax to output to
- **CMB_unit** – scale results from dimensionless. Use ‘muK’ for μK^2 units for CMB C_ℓ
- **raw_cl** – return C_ℓ rather than $\ell(\ell + 1)C_\ell/2\pi$

Returns numpy array CL[0:lmax+1,0:4], where 0..3 indexes TT, EE, BB, TE.

get_linear_matter_power_spectrum (*var1=None, var2=None, hubble_units=True, k_hunit=True, have_power_spectra=True, params=None, nonlinear=False*)

Calculates $P_{xy}(k)$, where x, y are one of model.Transfer_cdm, model.Transfer_xx etc. The output k values are not regularly spaced, and not interpolated. They are either k or k/h depending on the value of k_hunit (default True gives k/h).

For a description of outputs for different var1, var2 see [Matter power spectrum and matter transfer function variables](#).

Parameters

- **var1** – variable i (index, or name of variable; default delta_tot)
- **var2** – variable j (index, or name of variable; default delta_tot)
- **hubble_units** – if true, output power spectrum in (Mpc/h) units, otherwise Mpc
- **k_hunit** – if true, matter power is a function of k/h, if false, just k (both Mpc^{-1} units)
- **have_power_spectra** – set to False if not already computed power spectra
- **params** – if have_power_spectra=False, optional *CAMBparams* instance to specify new parameters
- **nonlinear** – include non-linear correction from halo model

Returns k/h or k, z, PK, where kz an z are arrays of k/h or k and z respectively, and PK[i,j] is the value at z[i], k[j]/h or k[j]

get_matter_power_interpolator (*nonlinear=True, var1=None, var2=None, hubble_units=True, k_hunit=True, return_z_k=False, log_interp=True, extrap_kmax=None, silent=False*)

Assuming transfers have been calculated, return a 2D spline interpolation object to evaluate matter power spectrum as function of z and k/h (or k). Uses self.Params.Transfer.PK_redshifts as the spline node points

in z . If fewer than four redshift points are used the interpolator uses a reduced order spline in z (so results at intermediate z may be inaccurate), otherwise it uses bicubic. Usage example:

```
PK = results.get_matter_power_interpolator();
print('Power spectrum at z=0.5, k/h=0.1 is %s (Mpc/h)^3'%(PK.P(0.5, 0.1)))
```

For a description of outputs for different var1, var2 see *Matter power spectrum and matter transfer function variables*.

Parameters

- **nonlinear** – include non-linear correction from halo model
- **var1** – variable i (index, or name of variable; default delta_tot)
- **var2** – variable j (index, or name of variable; default delta_tot)
- **hubble_units** – if true, output power spectrum in $(\text{Mpc}/h)^3$ units, otherwise Mpc^3
- **k_hunit** – if true, matter power is a function of k/h , if false, just k (both Mpc^{-1} units)
- **return_z_k** – if true, return interpolator, z , k where z , k are the grid used
- **log_interp** – if true, interpolate log of power spectrum (unless any values cross zero in which case ignored)
- **extrap_kmax** – if set, use power law extrapolation beyond k_{max} to $extrap_k_{\text{max}}$ (useful for tails of integrals)
- **silent** – Set True to silence warnings

Returns An object PK based on `RectBivariateSpline`, that can be called with `PK.P(z,kh)` or `PK(z,log(kh))` to get log matter power values. If `return_z_k=True`, instead return interpolator, z , k where z , k are the grid used.

`get_matter_power_spectrum` (*minkh=0.0001, maxkh=1.0, npoints=100, var1=None, var2=None, have_power_spectra=False, params=None*)

Calculates $P_{xy}(k/h)$, where x , y are one of `Transfer_cdm`, `Transfer_xx` etc. The output k values are regularly log spaced and interpolated. If `NonLinear` is set, the result is non-linear.

For a description of outputs for different var1, var2 see *Matter power spectrum and matter transfer function variables*.

Parameters

- **minkh** – minimum value of k/h for output grid (very low values $< 1e-4$ may not be calculated)
- **maxkh** – maximum value of k/h (check consistent with input `params.Transfer.kmax`)
- **npoints** – number of points equally spaced in log k
- **var1** – variable i (index, or name of variable; default delta_tot)
- **var2** – variable j (index, or name of variable; default delta_tot)
- **have_power_spectra** – set to True if already computed power spectra
- **params** – if `have_power_spectra=False` and want to specify new parameters, a `CAMBparams` instance

Returns kh , z , PK, where kh and z are arrays of k/h and z respectively, and `PK[i,j]` is value at $z[i]$, $kh[j]$

`get_matter_transfer_data` () → `camb.results.MatterTransferData`
Get matter transfer function data and `sigma8` for calculated results.

Returns *MatterTransferData* instance holding output arrays (copies, not pointers)

get_nonlinear_matter_power_spectrum (*var1=None, var2=None, hubble_units=True, k_hunit=True, have_power_spectra=True, params=None*)

Calculates $P_{xy}(k/h)$, where x, y are one of model.Transfer_cdm, model.Transfer_xx etc. The output k values are not regularly spaced, and not interpolated.

For a description of outputs for different var1, var2 see *Matter power spectrum and matter transfer function variables*.

Parameters

- **var1** – variable i (index, or name of variable; default delta_tot)
- **var2** – variable j (index, or name of variable; default delta_tot)
- **hubble_units** – if true, output power spectrum in $(\text{Mpc}/h)^3$ units, otherwise Mpc^3
- **k_hunit** – if true, matter power is a function of k/h, if false, just k (both Mpc^{-1} units)
- **have_power_spectra** – set to False if not already computed power spectra
- **params** – if have_power_spectra=False, optional *CAMBparams* instance to specify new parameters

Returns k/h or k, z, PK, where kz an z are arrays of k/h or k and z respectively, and PK[i,j] is the value at z[i], k[j]/h or k[j]

get_redshift_evolution (*q, z, vars=['k/h', 'delta_cdm', 'delta_baryon', 'delta_photon', 'delta_neutrino', 'delta_nu', 'delta_tot', 'delta_nonu', 'delta_tot_de', 'Weyl', 'v_newtonian_cdm', 'v_newtonian_baryon', 'v_baryon_cdm', 'a', 'etak', 'H', 'growth', 'v_photon', 'pi_photon', 'E_2', 'v_neutrino', 'T_source', 'E_source', 'lens_potential_source'], lAccuracyBoost=4*)

Get the mode evolution as a function of redshift for some k values.

Parameters

- **q** – wavenumber values to calculate (or array of k values)
- **z** – array of redshifts to output
- **vars** – list of variable names or camb.symbolic sympy expressions to output
- **lAccuracyBoost** – boost factor for ell accuracy (e.g. to get nice smooth curves for plotting)

Returns nd array, $A_{\{qti\}}$, size(q) x size(times) x len(vars), or 2d array if q is scalar

get_sigma8 ()

Get σ_8 values at Params.PK_redshifts (must previously have calculated power spectra)

Returns array of σ_8 values, in order of increasing time (decreasing redshift)

get_sigma8_0 ()

Get σ_8 value today (must previously have calculated power spectra)

Returns σ_8 today

get_sigmaR (*R, z_indices=None, var1=None, var2=None, hubble_units=True, return_R_z=False*)

Calculate σ_R values, the RMS linear matter fluctuation in spheres of radius R in linear theory. Accuracy depends on the sampling with which the matter transfer functions are computed.

For a description of outputs for different var1, var2 see *Matter power spectrum and matter transfer function variables*. Note that numerical errors are slightly different to get_sigma8 for R=8 Mpc/h.

Parameters

- **R** – radius in Mpc or h^{-1} Mpc units, scalar or array
- **z_indices** – indices of redshifts in Params.Transfer.PK_redshifts to calculate (default None gives all computed in order of increasing time (decreasing redshift); -1 gives redshift 0; list gives all listed indices)
- **var1** – variable i (index, or name of variable; default delta_tot)
- **var2** – variable j (index, or name of variable; default delta_tot)
- **hubble_units** – if true, R is in h^{-1} Mpc, otherwise Mpc
- **return_R_z** – if true, return tuple of R, z, sigmaR (where R always Mpc units not h^{-1} Mpc and $R < z$ are arrays)

Returns array of σ_R values, or 2D array indexed by (redshift, R)

get_source_cls_dict (*params=None, lmax=None, raw_cl=False*)

Get all source window function and CMB lensing and cross power spectra. Does not include CMB spectra. Note that P is the deflection angle, but lensing windows return the kappa power.

Parameters

- **params** – optional *CAMBparams* instance with parameters to use. If None, must have previously set parameters and called *calc_power_spectra()* (e.g. if you got this instance using *camb.get_results()*),
- **lmax** – maximum ℓ
- **raw_cl** – return C_ℓ rather than $\ell(\ell + 1)C_\ell/2\pi$

Returns dictionary of power spectrum arrays, index as PXP, PxW1, W1xW2, ... etc.

get_tensor_cls (*lmax=None, CMB_unit=None, raw_cl=False*)

Get tensor CMB power spectra. Must have already calculated power spectra.

Parameters

- **lmax** – lmax to output to
- **CMB_unit** – scale results from dimensionless. Use 'muK' for μK^2 units for CMB C_ℓ
- **raw_cl** – return C_ℓ rather than $\ell(\ell + 1)C_\ell/2\pi$

Returns numpy array CL[0:lmax+1,0:4], where 0..3 indexes TT, EE, BB, TE

get_time_evolution (*q, eta, vars=['k/h', 'delta_cdm', 'delta_baryon', 'delta_photon', 'delta_neutrino', 'delta_nu', 'delta_tot', 'delta_nonu', 'delta_tot_de', 'Weyl', 'v_newtonian_cdm', 'v_newtonian_baryon', 'v_baryon_cdm', 'a', 'etak', 'H', 'growth', 'v_photon', 'pi_photon', 'E_2', 'v_neutrino', 'T_source', 'E_source', 'lens_potential_source'], lAccuracyBoost=4, frame='CDM'*)

Get the mode evolution as a function of conformal time for some k values.

Parameters

- **q** – wavenumber values to calculate (or array of k values)
- **eta** – array of requested conformal times to output
- **vars** – list of variable names or sympy symbolic expressions to output (using *camb.symbolic*)
- **lAccuracyBoost** – factor by which to increase l_max in hierarchies compared to default - often needed to get nice smooth curves of acoustic oscillations for plotting.

- **frame** – for symbolic expressions, can specify frame name if the variable is not gauge invariant. e.g. specifying `Delta_g` and `frame='Newtonian'` would give the Newtonian gauge photon density perturbation.

Returns nd array, $A_{\{q\}}$, size(q) x size(times) x len(vars), or 2d array if q is scalar

get_total_cls (*lmax=None, CMB_unit=None, raw_cl=False*)

Get lensed-scalar + tensor CMB power spectra. Must have already calculated power spectra.

Parameters

- **lmax** – lmax to output to
- **CMB_unit** – scale results from dimensionless. Use ‘muK’ for μK^2 units for CMB C_ℓ
- **raw_cl** – return C_ℓ rather than $\ell(\ell + 1)C_\ell/2\pi$

Returns numpy array $CL[0:lmax+1,0:4]$, where 0..3 indexes TT, EE, BB, TE

get_unlensed_scalar_array_cls (*lmax=None*)

Get array of all cross power spectra. Must have already calculated power spectra. Results are dimensionless, and not scaled by `custom_scaled_ell_fac`.

Parameters **lmax** – lmax to output to

Returns numpy array $CL[0:, 0:0:lmax+1]$, where 0.. index T, E, lensing potential, source window functions

get_unlensed_scalar_cls (*lmax=None, CMB_unit=None, raw_cl=False*)

Get unlensed scalar CMB power spectra. Must have already calculated power spectra.

Parameters

- **lmax** – lmax to output to
- **CMB_unit** – scale results from dimensionless. Use ‘muK’ for μK^2 units for CMB C_ℓ
- **raw_cl** – return C_ℓ rather than $\ell(\ell + 1)C_\ell/2\pi$

Returns numpy array $CL[0:lmax+1,0:4]$, where 0..3 indexes TT, EE, BB, TE. $CL[:,2]$ will be zero.

get_unlensed_total_cls (*lmax=None, CMB_unit=None, raw_cl=False*)

Get unlensed CMB power spectra, including tensors if relevant. Must have already calculated power spectra.

Parameters

- **lmax** – lmax to output to
- **CMB_unit** – scale results from dimensionless. Use ‘muK’ for μK^2 units for CMB C_ℓ
- **raw_cl** – return C_ℓ rather than $\ell(\ell + 1)C_\ell/2\pi$

Returns numpy array $CL[0:lmax+1,0:4]$, where 0..3 indexes TT, EE, BB, TE.

h_of_z (z)

Get Hubble rate at redshift z, in Mpc^{-1} units, scalar or array

Must have called `calc_background()`, `calc_background_no_thermo()` or calculated transfer functions or power spectra.

Use `hubble_parameter` instead if you want in [km/s/Mpc] units.

Parameters **z** – redshift

Returns $H(z)$

hubble_parameter (*z*)

Get Hubble rate at redshift *z*, in km/s/Mpc units. Scalar or array.

Must have called `calc_background()`, `calc_background_no_thermo()` or calculated transfer functions or power spectra.

Parameters *z* – redshift

Returns $H(z)$ /[km/s/Mpc]

luminosity_distance (*z*)

Get luminosity distance from to redshift *z*.

Must have called `calc_background()`, `calc_background_no_thermo()` or calculated transfer functions or power spectra.

Parameters *z* – redshift or array of redshifts

Returns luminosity distance (matches rank of *z*)

physical_time (*z*)

Get physical time from hot big bang to redshift *z* in Gigayears.

Parameters *z* – redshift

Returns $t(z)$ /Gigayear

physical_time_a1_a2 (*a1*, *a2*)

Get physical time between two scalar factors in Gigayears

Must have called `calc_background()`, `calc_background_no_thermo()` or calculated transfer functions or power spectra.

Parameters

- **a1** – scale factor 1
- **a2** – scale factor 2

Returns $(\text{age}(a2) - \text{age}(a1))$ /Gigayear

power_spectra_from_transfer (*initial_power_params=None*, *silent=False*)

Assuming `calc_transfers()` or `calc_power_spectra()` have already been used, re-calculate the power spectra using a new set of initial power spectrum parameters with otherwise the same cosmology. This is typically much faster than re-calculating everything, as the transfer functions can be re-used. NOTE: if non-linear lensing is on, the transfer functions have the non-linear correction included when they are calculated, so using this function with a different initial power spectrum will not give quite the same results as doing a full recalculation.

Parameters

- **initial_power_params** – `initialpower.InitialPowerLaw` or `initialpower.SplinedInitialPower` instance with new primordial power spectrum parameters, or `None` to use current power spectrum.
- **silent** – suppress warnings about non-linear corrections not being recalculated

redshift_at_comoving_radial_distance (*chi*)

Convert comoving radial distance array to redshift array.

Parameters *chi* – comoving radial distance (in Mpc), scalar or array

Returns redshift at *chi*, scalar or array

redshift_at_conformal_time (*eta*)

Convert conformal time array to redshift array. Note that this function requires the transfers or background to have been calculated with `no_thermo=False` (the default).

Parameters *eta* – conformal time from big bang (in Mpc), scalar or array

Returns redshift at *eta*, scalar or array

replace (*instance*)

Replace the content of this class with another instance, doing a deep copy (in Fortran)

Parameters *instance* – instance of the same class to replace this instance with

save_cmb_power_spectra (*filename*, *lmax=None*, *CMB_unit='muK'*)

Save CMB power to a plain text file. Output is lensed total $\ell(\ell+1)C_\ell/2\pi$ then lensing potential and cross: L TT EE BB TE PP PT PE.

Parameters

- **filename** – filename to save
- **lmax** – lmax to save
- **CMB_unit** – scale results from dimensionless. Use 'muK' for μK^2 units for CMB C_ℓ and μK units for lensing cross.

set_params (*params*)

Set parameters from *params*. Note that this does not recompute anything; you will need to call `calc_transfers()` if you change any parameters affecting the background cosmology or the transfer function settings.

Parameters *params* – a *CAMBparams* instance

sound_horizon (*z*)

Get comoving sound horizon as function of redshift in Megaparsecs, the integral of the sound speed up to given redshift.

Parameters *z* – redshift or array of redshifts

Returns `r_s(z)`

class `camb.results.MatterTransferData`

`MatterTransferData` is the base class for storing matter power transfer function data for various *q* values. In a flat universe $q=k$, in a closed universe q is quantized.

To get an instance of this data, call `results.CAMBdata.get_matter_transfer_data()`.

For a description of the different `Transfer_XXX` outputs (and 21cm case) see *Matter power spectrum and matter transfer function variables*; the array is indexed by `index+1` given by:

- `Transfer_kh = 1` (k/h)
- `Transfer_cdm = 2` (cdm)
- `Transfer_b = 3` (baryons)
- `Transfer_g = 4` (photons)
- `Transfer_r = 5` (massless neutrinos)
- `Transfer_nu = 6` (massive neutrinos)
- `Transfer_tot = 7` (total matter)
- `Transfer_nonu = 8` (total matter excluding neutrinos)
- `Transfer_tot_de = 9` (total including dark energy perturbations)

- `Transfer_Weyl = 10` (Weyl potential)
- `Transfer_Newt_vel_cdm = 11` (Newtonian CDM velocity)
- `Transfer_Newt_vel_baryon = 12` (Newtonian baryon velocity)
- `Transfer_vel_baryon_cdm = 13` (relative baryon-cdm velocity)

Variables

- **nq** – number of q modes calculated
- **q** – array of q values calculated
- **sigma_8** – array of σ_8 values for each redshift
- **sigma2_vdelta_8** – array of v-delta8 correlation, so `sigma2_vdelta_8/sigma_8` can define growth
- **transfer_data** – numpy array T[entry, q_index, z_index] storing transfer functions for each redshift and q; entry+1 can be one of the `Transfer_xxx` variables above.

transfer_z (*name, z_index=0*)

Get transfer function (function of q, for each q in `self.q_trans`) by name for given redshift index

Parameters

- **name** – parameter name
- **z_index** – which redshift

Returns array of transfer function values for each calculated k

class `camb.results.CITransferData`

`CITransferData` is the base class for storing CMB power transfer functions, as a function of q and ℓ . To get an instance of this data, call `results.CAMBdata.get_cmb_transfer_data()`

Variables

- **NumSources** – number of sources calculated (size of p index)
- **q** – array of q values calculated (=k in flat universe)
- **L** – int array of ℓ values calculated
- **delta_p_l_k** – transfer functions, indexed by source, L, q

get_transfer (*source=0*)

Return C_ℓ transfer functions as a function of ℓ and q (= k in a flat universe).

Parameters **source** – index of source: e.g. 0 for temperature, 1 for E polarization, 2 for lensing potential

Returns array of computed L, array of computed q, transfer functions T(L,q)

Symbolic manipulation

This module defines the scalar linear perturbation equations for standard LCDM cosmology, using sympy. It uses the covariant perturbation notation, but includes functions to project into the Newtonian or synchronous gauge, as well as constructing general gauge invariant quantities. It uses “t” as the conformal time variable (=tau in the fortran code).

For a guide to usage and content see the [ScalEqs notebook](#)

As well as defining standard quantities, and how they map to CAMB variables, there are also functions for converting a symbolic expression to CAMB source code, and compiling custom sources for use with CAMB (as used by `model.CAMBparams.set_custom_scalar_sources()`, `results.CAMBdata.get_time_evolution()`)

A Lewis July 2017

`camb.symbolic.LinearPerturbation` (*name*, *species=None*, *camb_var=None*, *camb_sub=None*,
frame_dependence=None, *description=None*)

Returns as linear perturbation variable, a function of conformal time t. Use `help(x)` to quickly view all quantities defined for the result.

Parameters

- **name** – sympy name for the Function
- **species** – tag for the species if relevant (not used)
- **camb_var** – relevant CAMB fortran variable
- **camb_sub** – if not equal to `camb_var`, and string giving the expression in CAMB variables
- **frame_dependence** – the change in the perturbation when the frame 4-velocity u change from u to $u + \delta u$. Should be a sympy expression involving δu .
- **description** – string describing variable

Returns sympy Function instance (function of t), with attributes set to the arguments above.

`camb.symbolic.camb_fortran` (*expr*, *name='camb_function'*, *frame='CDM'*, *expand=False*)

Convert symbolic expression to CAMB fortran code, using CAMB variable notation. This is not completely general, but it will handle conversion of Newtonian gauge variables like Ψ_N , and most derivatives up to second order.

Parameters

- **expr** – symbolic sympy expression using camb.symbolic variables and functions (plus any standard general functions that CAMB can convert to fortran).
- **name** – lhs variable string to assign result to
- **frame** – frame in which to interpret non gauge-invariant expressions. By default uses CDM frame (synchronous gauge), as used natively by CAMB.
- **expand** – do a sympy expand before generating code

Returns fortran code snippet

`camb.symbolic.cdm_gauge(x)`

Evaluates an expression in the CDM frame ($v_c = 0, A = 0$). Equivalent to the synchronous gauge but using the covariant variable names.

Parameters **x** – expression

Returns expression evaluated in CDM frame.

`camb.symbolic.compile_source_function_code(code_body, file_path="", compiler=None, fflags=None, cache=True)`

Compile fortran code into function pointer in compiled shared library. The function is not intended to be called from python, but for passing back to compiled CAMB.

Parameters

- **code_body** – fortran code to do calculation and assign sources(i) output array. Can start with declarations of temporary variables if needed.
- **file_path** – optional output path for generated f90 code
- **compiler** – compiler, usually on path
- **fflags** – options for compiler
- **cache** – whether to cache the result

Returns function pointer for compiled code

class `camb.symbolic.f_K`

`camb.symbolic.get_hierarchies(lmax=5)`

Get Boltzmann hierarchies up to lmax for photons (J), E polarization and massless neutrinos (G).

Parameters **lmax** – maximum multipole

Returns list of equations

`camb.symbolic.get_scalar_temperature_sources(checks=False)`

Derives terms in line of sight source, after integration by parts so that only integrated against a Bessel function (no derivatives).

Parameters **checks** – True to do consistency checks on result

Returns monopole_source, ISW, doppler, quadrupole_source

`camb.symbolic.make_frame_invariant(expr, frame='CDM')`

Makes the quantity gauge invariant, assuming currently evaluated in frame 'frame'. frame can either be a string frame name, or a variable that is zero in the current frame,

e.g. frame = Delta_g gives the constant photon density frame. So make_frame_invariant(sigma, frame=Delta_g) will return the combination of sigma and Delta_g that is frame invariant (and equal to just sigma when Delta_g=0).

`camb.symbolic.newtonian_gauge(x)`

Evaluates an expression in the Newtonian gauge (zero shear, $\sigma=0$). Converts to using conventional metric perturbation variables for metric

$$ds^2 = a^2 \left((1 + 2\Psi_N) dt^2 - (1 - 2\Phi_N) \delta_{ij} dx^i dx^j \right)$$

Parameters **x** – expression

Returns expression evaluated in the Newtonian gauge

`camb.symbolic.synchronous_gauge(x)`

evaluates an expression in the synchronous gauge, using conventional synchronous-gauge variables.

Parameters **x** – expression

Returns synchronous gauge variable expression

Other modules:

class `camb.bbn.BBNInterpolator` (*x, y, z, bbox=[None, None, None, None], kx=3, ky=3, s=0*)

class `camb.bbn.BBNPredictor`

The base class for making BBN predictions for Helium abundance

Y_He (*ombh2, delta_neff=0.0*)

Get BBN helium mass fraction for CMB code.

Parameters

- **ombh2** – $\Omega_b h^2$
- **delta_neff** – additional N_{eff} relative to standard value (of 3.046)

Returns Y_He helium mass fraction predicted by BBN

Y_p (*ombh2, delta_neff=0.0*)

Get BBN helium nucleon fraction. Must be implemented by extensions.

Parameters

- **ombh2** – $\Omega_b h^2$
- **delta_neff** – additional N_{eff} relative to standard value (of 3.046)

Returns Y_p helium nucleon fraction predicted by BBN

class `camb.bbn.BBN_fitting_parthenope` (*tau_neutron=None*)

Old BBN predictions for Helium abundance using fitting formulae based on Parthenope (pre 2015).

Y_p (*ombh2, delta_neff=0.0, tau_neutron=None*)

Get BBN helium nucleon fraction. # Parthenope fits, as in Planck 2015 papers

Parameters

- **ombh2** – $\Omega_b h^2$
- **delta_neff** – additional N_{eff} relative to standard value (of 3.046)
- **tau_neutron** – neutron lifetime

Returns Y_p^{BBN} helium nucleon fraction predicted by BBN

class `camb.bbn.BBN_table_interpolator` (*interpolation_table*='ParthENoPE_880.2_standard.dat',
function_of=('ombh2', 'DeltaN'))

BBN predictor based on interpolation from a numerical table calculated by a BBN code.

Tables are supplied for [Parthenope 2017](#) (ParthENoPE_880.2_standard.dat, default), similar but with Marucci rates (ParthENoPE_880.2_marcucci.dat), and [PRIMAT](#) (PRIMAT_Yp_DH_Error.dat).

Parameters

- **interpolation_table** – filename of interpolation table to use.
- **function_of** – two variables that determine the interpolation grid (x,y) in the table, matching top column label comment. By default ombh2, DeltaN, and function argument names reflect that, but can also be used more generally.

DH (*ombh2, delta_neff=0.0, grid=False*)

Get deuterium ratio D/H by interpolation in table

Parameters

- **ombh2** – $\Omega_b h^2$ (or, more generally, value of function_of[0])
- **delta_neff** – additional N_{eff} relative to standard value (of 3.046) (or value of function_of[1])
- **grid** – parameter for `RectBivariateSpline` (whether to evaluate the results on a grid spanned by the input arrays, or at points specified by the input arrays)

Returns D/H

Y_p (*ombh2, delta_neff=0.0, grid=False*)

Get BBN helium nucleon fraction by interpolation in table.

Parameters

- **ombh2** – $\Omega_b h^2$ (or, more generally, value of function_of[0])
- **delta_neff** – additional N_{eff} relative to standard value (of 3.046) (or value of function_of[1])
- **grid** – parameter for `RectBivariateSpline` (whether to evaluate the results on a grid spanned by the input arrays, or at points specified by the input arrays)

Returns Y_p helium nucleon fraction predicted by BBN. Call `Y_He()` to get mass fraction instead.

get (*name, ombh2, delta_neff=0.0, grid=False*)

Get value for variable “name” by interpolation from table (where name is given in the column header comment) For example `get('sig(D/H)',0.0222,0)` to get the error on D/H

Parameters

- **name** – string name of the parameter, as given in header of interpolation table
- **ombh2** – $\Omega_b h^2$ (or, more generally, value of function_of[0])
- **delta_neff** – additional N_{eff} relative to standard value (of 3.046) (or value of function_of[1])
- **grid** – parameter for `RectBivariateSpline` (whether to evaluate the results on a grid spanned by the input arrays, or at points specified by the input arrays)

Returns Interpolated value (or grid)

`camb.bbn.get_predictor` (*predictor_name=None*)

Get instance of default BBNPredictor class. Currently numerical table interpolation as Planck 2018 analysis.

Dark Energy models

class `camb.dark_energy.DarkEnergyModel`
 Abstract base class for dark energy model implementations.

class `camb.dark_energy.DarkEnergyEqnOfState`
 Bases: `camb.dark_energy.DarkEnergyModel`

Abstract base class for models using w and w_a parameterization with use $w(a) = w + (1-a)*w_a$ parameterization, or call `set_w_a_table` to set another tabulated $w(a)$. If tabulated $w(a)$ is used, w and w_a are set to approximate values at $z=0$.

See `model.CAMBparams.set_initial_power_function()` for a convenience constructor function to set a general interpolated $P(k)$ model from a python function.

Variables

- **w** – (*float64*) $w(0)$
- **wa** – (*float64*) $-dw/da(0)$
- **cs2** – (*float64*) fluid rest-frame sound speed squared
- **use_tabulated_w** – (*boolean*) using an interpolated tabulated $w(a)$ rather than w , w_a above

set_params ($w=-1.0, wa=0, cs2=1.0$)

Set the parameters so that $P(a)/\rho(a) = w(a) = w + (1-a)*w_a$

Parameters

- **w** – $w(0)$
- **wa** – $-dw/da(0)$
- **cs2** – fluid rest-frame sound speed squared

set_w_a_table (a, w)

Set $w(a)$ from numerical values (used as cubic spline). Note this is quite slow.

Parameters

- **a** – array of scale factors
- **w** – array of $w(a)$

Returns self**class** `camb.dark_energy.DarkEnergyFluid`Bases: `camb.dark_energy.DarkEnergyEqnOfState`

Class implementing the w , w_a or splined $w(a)$ parameterization using the constant sound-speed single fluid model (as for single-field quintessence).

class `camb.dark_energy.DarkEnergyPPF`Bases: `camb.dark_energy.DarkEnergyEqnOfState`

Class implementing the w , w_a or splined $w(a)$ parameterization in the PPF perturbation approximation ([arXiv:0808.3125](#)) Use inherited methods to set parameters or interpolation table.

class `camb.dark_energy.AxionEffectiveFluid`Bases: `camb.dark_energy.DarkEnergyModel`

Example implementation of a specific (early) dark energy fluid model ([arXiv:1806.10608](#)). Not well tested, but should serve to demonstrate how to make your own custom classes.

Variables

- **w_n** – (*float64*)
- **om** – (*float64*)
- **a_c** – (*float64*)
- **theta_i** – (*float64*)

Initial power spectra

class `camb.initialpower.InitialPower`

Abstract base class for initial power spectrum classes

class `camb.initialpower.InitialPowerLaw` (**kwargs)

Bases: `camb.initialpower.InitialPower`

Object to store parameters for the primordial power spectrum in the standard power law expansion.

Variables

- **tensor_parameterization** – (integer/string, one of: `tensor_param_indeptilt`, `tensor_param_rpivot`, `tensor_param_AT`)
- **ns** – (*float64*)
- **nrun** – (*float64*)
- **nrunrun** – (*float64*)
- **nt** – (*float64*)
- **ntrun** – (*float64*)
- **r** – (*float64*)
- **pivot_scalar** – (*float64*)
- **pivot_tensor** – (*float64*)
- **As** – (*float64*)
- **At** – (*float64*)

has_tensors ()

Do these settings have non-zero tensors?

Returns True if non-zero tensor amplitude

set_params (*As=2e-09, ns=0.96, nrun=0, nrunrun=0.0, r=0.0, nt=None, ntrun=0.0, pivot_scalar=0.05, pivot_tensor=0.05, parameterization='tensor_param_rpivot'*)

Set parameters using standard power law parameterization. If `nt=None`, uses inflation consistency relation.

Parameters

- **As** – comoving curvature power at $k=\text{pivot_scalar}$ (A_s)
- **ns** – scalar spectral index n_s
- **nrun** – running of scalar spectral index $dn_s/d\log k$
- **nrunrun** – running of running of spectral index, $d^2n_s/d(\log k)^2$
- **r** – tensor to scalar ratio at pivot
- **nt** – tensor spectral index n_t . If None, set using inflation consistency
- **ntrun** – running of tensor spectral index
- **pivot_scalar** – pivot scale for scalar spectrum
- **pivot_tensor** – pivot scale for tensor spectrum
- **parameterization** – See CAMB notes. One of - tensor_param_indeptilt = 1 - tensor_param_rpivot = 2 - tensor_param_AT = 3

Returns self

class camb.initialpower.SplinedInitialPower (**kwargs)

Bases: *camb.initialpower.InitialPower*

Object to store a generic primordial spectrum set from a set of sampled k_i , $P(k_i)$ values

Variables **effective_ns_for_nonlinear** – (*float64*) Effective n_s to use for approximate non-linear correction models

has_tensors ()

Is the tensor spectrum set?

Returns True if tensors

set_scalar_log_regular (*kmin, kmax, PK*)

Set log-regular cubic spline interpolation for $P(k)$

Parameters

- **kmin** – minimum k value (not minimum $\log(k)$)
- **kmax** – maximum k value (inclusive)
- **PK** – array of scalar power spectrum values, with $PK[0]=P(kmin)$ and $PK[-1]=P(kmax)$

set_scalar_table (*k, PK*)

Set arrays of k and $P(k)$ values for cubic spline interpolation. Note that using *set_scalar_log_regular()* may be better (faster, and easier to get fine enough spacing a low k)

Parameters

- **k** – array of k values (Mpc^{-1})
- **PK** – array of scalar power spectrum values

set_tensor_log_regular (*kmin, kmax, PK*)

Set log-regular cubic spline interpolation for tensor spectrum $P_t(k)$

Parameters

- **kmin** – minimum k value (not minimum $\log(k)$)
- **kmax** – maximum k value (inclusive)

- **PK** – array of scalar power spectrum values, with $PK[0]=P_t(k_{\min})$ and $PK[-1]=P_t(k_{\max})$

set_tensor_table (*k*, *PK*)

Set arrays of *k* and $P_t(k)$ values for cubic spline interpolation

Parameters

- **k** – array of *k* values (Mpc^{-1})
- **PK** – array of tensor power spectrum values

Non-linear models

class `camb.nonlinear.NonLinearModel`

Abstract base class for non-linear correction models

Variables `Min_kh_nonlinear` – (*float64*) minimum k/h at which to apply non-linear corrections

class `camb.nonlinear.Halofit`

Bases: `camb.nonlinear.NonLinearModel`

Various specific approximate non-linear correction models based on HaloFit.

Variables

- `halofit_version` – (integer/string, one of: original, bird, peacock, takahashi, mead, halomodel, casarini, mead2015)
- `HMCode_A_baryon` – (*float64*) HMCode parameter A_baryon
- `HMCode_eta_baryon` – (*float64*) HMCode parameter eta_baryon

set_params (`halofit_version='mead', HMCode_A_baryon=3.13, HMCode_eta_baryon=0.603`)

Set the halofit model for non-linear corrections.

Parameters

- `halofit_version` – One of
 - original: astro-ph/0207664
 - bird: arXiv:1109.4416
 - peacock: Peacock fit
 - takahashi: arXiv:1208.2701
 - mead: HMCode arXiv:1602.02154
 - halomodel: basic halomodel
 - casarini: PKequal arXiv:0810.0190, arXiv:1601.07230
 - mead2015: original 2015 version of HMCode arXiv:1505.07833

- `HMCode_A_baryon` – HMCode parameter `A_baryon`. Default 3.13.
- `HMCode_eta_baryon` – HMCode parameter `eta_baryon`. Default 0.603.

class `camb.nonlinear.SecondOrderPK`

Bases: `camb.nonlinear.NonLinearModel`

Third-order Newtonian perturbation theory results for the non-linear correction. Only intended for use at very high redshift ($z > 10$) where corrections are perturbative, it will not give sensible results at low redshift.

See Appendix F of [astro-ph/0702600](https://arxiv.org/abs/astro-ph/0702600) for equations and references.

Not intended for production use, it's mainly to serve as an example alternative non-linear model implementation.

Reionization models

class `camb.reionization.ReionizationModel`

Abstract base class for reionization models.

Variables `Reionization` – (*boolean*) Is there reionization? (can be off for matter power which is independent of it)

class `camb.reionization.TanhReionization`

Bases: `camb.reionization.ReionizationModel`

This default (unphysical) tanh x_e parameterization is described in Appendix B of [arXiv:0804.3865](https://arxiv.org/abs/0804.3865)

Variables

- `use_optical_depth` – (*boolean*) Whether to use the optical depth or redshift parameters
- `redshift` – (*float64*) Reionization redshift if `use_optical_depth=False`
- `optical_depth` – (*float64*) Optical depth if `use_optical_depth=True`
- `delta_redshift` – (*float64*) Duration of reionization
- `fraction` – (*float64*) Reionization fraction when complete, or -1 for full ionization of hydrogen and first ionization of helium.
- `include_helium_fullreion` – (*boolean*) Whether to include second reionization of helium
- `helium_redshift` – (*float64*) Redshift for second reionization of helium
- `helium_delta_redshift` – (*float64*) Width in redshift for second reionization of helium
- `helium_redshiftstart` – (*float64*) Include second helium reionization below this redshift
- `tau_solve_accuracy_boost` – (*float64*) Accuracy boosting parameter for solving for z_{re} from τ
- `timestep_boost` – (*float64*) Accuracy boosting parameter for the minimum number of time sampling steps through reionization

- **max_redshift** – (*float64*) Maximum redshift allowed when mapping tau into reionization redshift

get_zre (*params, tau=None*)

Get the midpoint redshift of reionization.

Parameters

- **params** – *model.CAMBparams* instance with cosmological parameters
- **tau** – if set, calculate the redshift for optical depth tau, otherwise uses currently set parameters

Returns reionization mid-point redshift

set_tau (*tau, delta_redshift=None*)

Set the optical depth

Parameters

- **tau** – optical depth
- **delta_redshift** – delta z for reionization

Returns self

set_zrei (*zrei, delta_redshift=None*)

Set the mid-point reionization redshift

Parameters

- **zrei** – mid-point redshift
- **delta_redshift** – delta z for reionization

Returns self

Recombination models

class `camb.recombination.RecombinationModel`

Abstract base class for recombination models

Variables `min_a_evolve_Tm` – (*float64*) minimum scale factor at which to solve matter temperature perturbation if evolving sound speed or ionization fraction perturbations

class `camb.recombination.Recfast`

Bases: `camb.recombination.RecombinationModel`

RECFAST recombination model (see recfast source for details).

Variables

- `RECFAST_fudge` – (*float64*)
- `RECFAST_fudge_He` – (*float64*)
- `RECFAST_Heswitch` – (*integer*)
- `RECFAST_Hswitch` – (*boolean*)
- `AGauss1` – (*float64*)
- `AGauss2` – (*float64*)
- `zGauss1` – (*float64*)
- `zGauss2` – (*float64*)
- `wGauss1` – (*float64*)
- `wGauss2` – (*float64*)

class `camb.recombination.CosmoRec`

Bases: `camb.recombination.RecombinationModel`

`CosmoRec` recombination model. To use this, the library must be build with `CosmoRec` installed and `RECOMBINATION_FILES` including `cosmorec` in the Makefile.

`CosmoRec` must be built with `-fPIC` added to the compiler flags.

Variables

- **runmode** – (*integer*) Default 0, with diffusion; 1: without diffusion; 2: RECFAST++, 3: RECFAST++ run with correction
- **fdm** – (*float64*) Dark matter annihilation efficiency
- **accuracy** – (*float64*) 0-normal, 3-most accurate

class `camb.recombination.HyRec`

Bases: `camb.recombination.RecombinationModel`

`HyRec` recombination model. To use this, the library must be build with `HyRec` installed and `RECOMBINATION_FILES` including `hyrec` in the Makefile.

You will need to edit `HyRec` Makefile to add `-fPIC` compiler flag to `CCFLAG` (for `gcc`), and rename “`dtauda_`” in `history.c` to “`exported_dtauda`”

Source windows functions

class `camb.sources.SourceWindow`

Abstract base class for a number count/lensing/21cm source window function. A list of instances of these classes can be assigned to the `SourceWindows` field of `model.CAMBparams`.

Note that source windows can currently only be used in flat models.

Variables

- **source_type** – (integer/string, one of: 21cm, counts, lensing)
- **bias** – (*float64*)
- **dlog10Ndm** – (*float64*)

class `camb.sources.GaussianSourceWindow`

Bases: `camb.sources.SourceWindow`

A Gaussian $W(z)$ source window function.

Variables

- **redshift** – (*float64*)
- **sigma** – (*float64*)

class `camb.sources.SplinedSourceWindow` (***kwargs*)

Bases: `camb.sources.SourceWindow`

A numerical $W(z)$ source window function constructed by interpolation from a numerical table.

set_table (*z*, *W*)

Set arrays of z and $W(z)$ for cubic spline interpolation. Note that $W(z)$ is the total count distribution observed, not a fractional selection function on an underlying distribution.

Parameters

- **z** – array of redshift values (monotonically increasing)
- **W** – array of window function values. It must be well enough sampled to smoothly cubic-spline interpolate

Correlation functions

Functions to transform CMB angular power spectra into correlation functions (`cl2corr`) and vice versa (`corr2cl`), and calculate lensed power spectra from unlensed ones.

The lensed power spectrum functions are not intended to replace those calculated by default when getting CAMB results, but may be useful for tests, e.g. using different lensing potential power spectra, partially-delensed lensing power spectra, etc.

These functions are all pure python/scipy, and operate and return `cls` including factors $\ell(\ell + 1)/2\pi$ (for CMB) and $[L(L + 1)]^2/2\pi$ (for lensing).

A. Lewis December 2016

`camb.correlations.cl2corr` (*cls*, *xvals*, *lmax=None*)

Get the correlation function from the power spectra, evaluated at points $\cos(\theta) = xvals$. Use roots of Legendre polynomials (`np.polynomial.legendre.leggauss`) for accurate back integration with `corr2cl`. Note currently does not work at `xvals=1` (can easily calculate that as special case!).

Parameters

- **cls** – 2D array `cls(L,ix)`, with $L (\equiv \ell)$ starting at zero and `ix=0,1,2,3` in order TT, EE, BB, TE. `cls` should include $\ell(\ell + 1)/2\pi$ factors.
- **xvals** – array of $\cos(\theta)$ values at which to calculate correlation function.
- **lmax** – optional maximum L to use from the `cls` arrays

Returns 2D array of `corrs[i, ix]`, where `ix=0,1,2,3` are T, Q+U, Q-U and cross

`camb.correlations.corr2cl` (*corrs*, *xvals*, *weights*, *lmax*)

Transform from correlation functions to power spectra. Note that using `cl2corr` followed by `corr2cl` is generally very accurate ($< 1e-5$ relative error) if `xvals`, `weights = np.polynomial.legendre.leggauss(lmax+1)`

Parameters

- **corrs** – 2D array, `corrs[i, ix]`, where `ix=0,1,2,3` are T, Q+U, Q-U and cross
- **xvals** – values of $\cos(\theta)$ at which `corrs` stores values

- **weights** – weights for integrating each point in `xvals`. Typically from `np.polynomial.legendre.leggauss`
- **lmax** – maximum ℓ to calculate C_ℓ

Returns array of power spectra, `cl[L, ix]`, where L starts at zero and `ix=0,1,2,3` in order TT, EE, BB, TE. They include $\ell(\ell + 1)/2\pi$ factors.

`camb.correlations.gauss_legendre_correlation` (*cls, lmax=None, sampling_factor=1*)

Transform power spectrum `cls` into correlation functions evaluated at the roots of the Legendre polynomials for Gauss-Legendre quadrature. Returns correlation function array, evaluation points and weights. Result can be passed to `corr2cl` for accurate back transform.

Parameters

- **cls** – 2D array `cls(L,ix)`, with L ($\equiv \ell$) starting at zero and `ix=0,1,2,3` in order TT, EE, BB, TE. Should include $\ell(\ell + 1)/2\pi$ factors.
- **lmax** – optional maximum L to use
- **sampling_factor** – uses Gauss-Legendre with degree `lmax*sampling_factor+1`

Returns `corrs, xvals, weights`; `corrs[i, ix]` is 2D array where `ix=0,1,2,3` are T, Q+U, Q-U and cross

`camb.correlations.legendre_funcs` (*lmax, x, m=(0, 2), lfacs=None, lfacs2=None, lroot-facs=None*)

Utility function to return array of Legendre and d_{mn} functions for all ℓ up to `lmax`. Note that d_{mn} arrays start at $\ell_{\min} = \max(m, n)$, so returned arrays are different sizes

Parameters

- **lmax** – maximum ℓ
- **x** – scalar value of $\cos(\theta)$ at which to evaluate
- **m** – m values to calculate $d_{m,n}$, etc as relevant
- **lfacs** – optional pre-computed $\ell(\ell + 1)$ float array
- **lfacs2** – optional pre-computed $(\ell + 2) * (\ell - 1)$ float array
- **lrootfacs** – optional pre-computed $\sqrt{\text{lfacs} * \text{lfacs2}}$ array

Returns $(P, P'), (d_{11}, d_{-1,1}), (d_{20}, d_{22}, d_{2,-2})$ as requested, where P starts at $\ell = 0$, but spin functions start at $\ell = \ell_{\min}$

`camb.correlations.lensed_cl_derivative_unlensed` (*clpp, lmax=None, theta_max=0.098174771875, apodize_point_width=10, sampling_factor=1.4*)

Get derivative `dcl` of lensed minus unlensed power $D_\ell \equiv \ell(\ell + 1)\Delta C_\ell/2\pi$ with respect to $\ell(\ell + 1)C_\ell^{\text{unlens}}/2\pi$

The difference in power in the lensed spectrum is given by `dCL[ix, :, :].dot(cl)`, where `cl` is the appropriate $\ell(\ell + 1)C_\ell^{\text{unlens}}/2\pi$.

Uses the non-perturbative curved-sky results from Eqs 9.12 and 9.16-9.18 of [astro-ph/0601594](#), to second order in $C_{\text{gl},2}$

Parameters

- **clpp** – array of $[L(L + 1)]^2 C_L^{\phi\phi}/2\pi$ lensing potential power spectrum (zero based)
- **lmax** – optional maximum L to use from the `clpp` array
- **theta_max** – maximum angle (in radians) to keep in the correlation functions

- **apodize_point_width** – if theta_max is set, apodize around the cut using half Gaussian of approx width apodize_point_width/lmax*pi
- **sampling_factor** – npoints = int(sampling_factor*lmax)+1

Returns array dCL[ix, ell, L], where ix=0,1,2,3 are TT, EE, BB, TE and result is $d(\Delta D_\ell^{\text{ix}})/dD_L^{\text{unlens},j}$ where j[ix] are TT, EE, EE, TE

```
camb.correlations.lensed_cl_derivatives (cls,          clpp,          lmax=None,
                                         theta_max=0.098174771875,
                                         apodize_point_width=10, sampling_factor=1.4)
```

Get derivative dcl of lensed $D_\ell \equiv \ell(\ell + 1)C_\ell/2\pi$ with respect to $\log(C_L^\phi)$. To leading order (and hence not actually accurate), the lensed correction to power spectrum ix is given by `dcl[ix,:].dot(np.ones(clpp.shape))`.

Uses the non-perturbative curved-sky results from Eqs 9.12 and 9.16-9.18 of [astro-ph/0601594](#), to second order in $C_{\text{gl},2}$

Parameters

- **cls** – 2D array of unlensed $\text{cls}(L, \text{ix})$, with L starting at zero and ix=0,1,2,3 in order TT, EE, BB, TE. cls should include $\ell(\ell + 1)/2\pi$ factors.
- **clpp** – array of $[L(L + 1)]^2 C_L^{\phi\phi}/2\pi$ lensing potential power spectrum (zero based)
- **lmax** – optional maximum L to use from the cls arrays
- **theta_max** – maximum angle (in radians) to keep in the correlation functions
- **apodize_point_width** – if theta_max is set, apodize around the cut using half Gaussian of approx width apodize_point_width/lmax*pi
- **sampling_factor** – npoints = int(sampling_factor*lmax)+1

Returns array dCL[ix, ell, L], where ix=0,1,2,3 are T, EE, BB, TE and r result is $d[D_\ell^{\text{ix}}]/d(\log C_L^\phi)$

```
camb.correlations.lensed_cls (cls,          clpp,          lmax=None,          lmax_lensed=None,
                              sampling_factor=1.4, delta_cls=False, theta_max=0.098174771875,
                              apodize_point_width=10, leggaus=True, cache=True)
```

Get the lensed power spectra from the unlensed power spectra and the lensing potential power. Uses the non-perturbative curved-sky results from Eqs 9.12 and 9.16-9.18 of [astro-ph/0601594](#), to second order in $C_{\text{gl},2}$.

Correlations are calculated for Gauss-Legendre integration if leggaus=True; this slows it by several seconds, but will be must faster on subsequent calls with the same lmax*sampling_factor. If Gauss-Legendre is not used, sampling_factor needs to be about 2 times larger for same accuracy.

For a reference implementation with the full integral range and no apodization set theta_max=None.

Note that this function does not pad high ℓ with a smooth fit (like CAMB's main functions); for accurate results should be called with lmax high enough that input cls are effectively band limited (lmax >= 2500, or higher for accurate BB to small scales). Usually lmax truncation errors are far larger than other numerical errors for lmax<4000.

Parameters

- **cls** – 2D array of unlensed $\text{cls}(L, \text{ix})$, with L starting at zero and ix=0,1,2,3 in order TT, EE, BB, TE. cls should include $\ell(\ell + 1)/2\pi$ factors.
- **clpp** – array of $[L(L + 1)]^2 C_L^{\phi\phi}/2\pi$ lensing potential power spectrum (zero based)
- **lmax** – optional maximum L to use from the cls arrays
- **lmax_lensed** – optional maximum L for the returned cl array (lmax_lensed <= lmax)
- **sampling_factor** – npoints = int(sampling_factor*lmax)+1

- **delta_cls** – if true, return the difference between lensed and unlensed (optional, default False)
- **theta_max** – maximum angle (in radians) to keep in the correlation functions; default: $\pi/32$
- **apodize_point_width** – if theta_max is set, apodize around the cut using half Gaussian of approx width $\text{apodize_point_width}/l_{\text{max}}*\pi$
- **leggaus** – whether to use Gauss-Legendre integration (default True)
- **cache** – if leggaus = True, set cache to save the x values and weights between calls (most of the time)

Returns 2D array of $\text{cls}[L, \text{ix}]$, with L starting at zero and $\text{ix}=0,1,2,3$ in order TT, EE, BB, TE. cls include $\ell(\ell + 1)/2\pi$ factors.

```
camb.correlations.lensed_correlations(cls, clpp, xvals, weights=None,
                                       lmax=None, delta=False, theta_max=None,
                                       apodize_point_width=10)
```

Get the lensed correlation function from the unlensed power spectra, evaluated at points $\cos(\theta) = \text{xvals}$. Use roots of Legendre polynomials (`np.polynomial.legendre.leggauss`) for accurate back integration with `corr2cl`. Note currently does not work at $\text{xvals}=1$ (can easily calculate that as special case!).

To get the lensed cls efficiently, set *weights* to the integral weights for each x value, then function returns lensed correlations and lensed cls .

Uses the non-perturbative curved-sky results from Eqs 9.12 and 9.16-9.18 of [astro-ph/0601594](#), to second order in $C_{\text{gl},2}$

Parameters

- **cls** – 2D array of unlensed $\text{cls}(L, \text{ix})$, with L ($\equiv \ell$) starting at zero and $\text{ix}=0,1,2,3$ in order TT, EE, BB, TE. cls should include $\ell(\ell + 1)/2\pi$ factors.
- **clpp** – array of $[L(L + 1)]^2 C_L^{\phi\phi}/2\pi$ lensing potential power spectrum (zero based)
- **xvals** – array of $\cos(\theta)$ values at which to calculate correlation function.
- **weights** – if given also return lensed C_ℓ , otherwise just lensed correlations
- **lmax** – optional maximum L to use from the cls arrays
- **delta** – if true, calculate the difference between lensed and unlensed (default False)
- **theta_max** – maximum angle (in radians) to keep in the correlation functions
- **apodize_point_width** – smoothing scale for apodization at truncation of correlation function

Returns 2D array of $\text{corrs}[i, \text{ix}]$, where $\text{ix}=0,1,2,3$ are T, Q+U, Q-U and cross; if *weights* is not None, then return corrs , lensed_cls

```
camb.correlations.lensing_R(clpp, lmax=None)
```

Get $R \equiv \frac{1}{2} \langle |\nabla\phi|^2 \rangle$

Parameters

- **clpp** – array of $[L(L + 1)]^2 C_L^{\phi\phi}/2\pi$ lensing potential power spectrum
- **lmax** – optional maximum L to use from the cls arrays

Returns R

```
camb.correlations.lensing_correlations(clpp, xvals, lmax=None)
```

Get the $\sigma^2(x)$ and $C_{\text{gl},2}(x)$ functions from the lensing power spectrum

Parameters

- **clpp** – array of $[L(L + 1)]^2 C_L^{\phi\phi} / 2\pi$ lensing potential power spectrum (zero based)
- **xvals** – array of $\cos(\theta)$ values at which to calculate correlation function.
- **lmax** – optional maximum L to use from the clpp array

Returns array of $\sigma^2(x)$, array of $C_{g1,2}(x)$

`camb.postborn.get_field_rotation_BB` (*params*, *lmax=None*, *acc=1*, *CMB_unit='muK'*,
raw_cl=False, *spline=True*)

Get the B-mode power spectrum from field post-born field rotation, based on perturbative and Limber approximations. See [arXiv:1605.05662](https://arxiv.org/abs/1605.05662).

Parameters

- **params** – *model.CAMBparams* instance with cosmological parameters etc.
- **lmax** – maximum ℓ
- **acc** – accuracy
- **CMB_unit** – units for CMB output relative to dimensionless
- **raw_cl** – return C_ℓ rather than $\ell(\ell + 1)C_\ell/2\pi$
- **spline** – return `InterpolatedUnivariateSpline`, otherwise return tuple of lists of ℓ and C_ℓ

Returns `InterpolatedUnivariateSpline` (or arrays of sampled ℓ and) $\ell^2 C_\ell^{BB}/(2\pi)$ (unless `raw_cl`, in which case just C_ℓ^{BB})

`camb.postborn.get_field_rotation_power` (*params*, *kmax=100*, *lmax=20000*, *non_linear=True*,
z_source=None, *k_per_logint=None*, *acc=1*,
lsamp=None)

Get field rotation power spectrum, $C_L^{\omega\omega}$, following [arXiv:1605.05662](https://arxiv.org/abs/1605.05662). Uses lowest Limber approximation.

Parameters

- **params** – *model.CAMBparams* instance with cosmological parameters etc.
- **kmax** – maximum k (in Mpc^{-1} units)
- **lmax** – maximum L
- **non_linear** – include non-linear corrections
- **z_source** – redshift of source. If `None`, use peak of CMB visibility for CMB lensing
- **k_per_logint** – sampling to use in k

- **acc** – accuracy setting, increase to test stability
- **lsamp** – array of L values to compute output at. If not set, set to sampling good for interpolation

Returns $L, C_L^{\omega\omega}$: the L sample values and corresponding rotation power

Lensing emission angle

This module calculates the corrections to the standard lensed CMB power spectra results due to time delay and emission angle, following [arXiv:1706.02673](https://arxiv.org/abs/1706.02673). This can be combined with the result from the postborn module to estimate the leading corrections to the standard lensing B modes.

Corrections to T and E are negligible, and not calculated. The result for BB includes approximately contributions from reionization, but this can optionally be turned off.

```
camb.emission_angle.get_emission_angle_powers(camb_background, PK, chi_source,
                                              lmax=3000, acc=1, lsamp=None)
```

Get the power spectrum of ψ_d , the potential for the emission angle, and its cross with standard lensing. Uses the Limber approximation (and assumes flat universe).

Parameters

- **camb_background** – a CAMB results object, used for calling background functions
- **PK** – a matter power spectrum interpolator (from `camb.get_matter_power_interpolator`)
- **chi_source** – comoving radial distance of source in Mpc
- **lmax** – maximum L
- **acc** – accuracy parameter
- **lsamp** – L sampling for the result

Returns a `InterpolatedUnivariateSpline` object containing $L(L + 1)C_L$

```
camb.emission_angle.get_emission_delay_BB(params, kmax=100, lmax=3000,
                                          non_linear=True, CMB_unit='muK',
                                          raw_cl=False, acc=1, lsamp=None,
                                          return_terms=False, include_reionization=True)
```

Get B modes from emission angle and time delay effects. Uses full-sky result from appendix of [arXiv:1706.02673](https://arxiv.org/abs/1706.02673)

Parameters

- **params** – `model.CAMBparams` instance with cosmological parameters etc.
- **kmax** – maximum k (in Mpc^{-1} units)

- **lmax** – maximum ℓ
- **non_linear** – include non-linear corrections
- **CMB_unit** – normalization for the result
- **raw_cl** – if true return C_ℓ , else $\ell(\ell + 1)C_\ell/2\pi$
- **acc** – accuracy setting, increase to test stability
- **lsamp** – array of ℓ values to compute output at. If not set, set to sampling good for interpolation
- **return_terms** – return the three sub-terms separately rather than the total
- **include_reionization** – approximately include reionization terms by second scattering surface

Returns InterpolatedUnivariateSpline for C_ℓ^{BB}

`camb.emission_angle.get_source_cmb_cl(params, CMB_unit='muK')`

Get the angular power spectrum of emission angle and time delay sources ψ_t , ψ_ζ , as well as the perpendicular velocity and E polarization. All are returned with 1 and 2 versions, for recombination and reionization respectively. Note that this function destroys any custom sources currently configured.

Parameters

- **params** – `model.CAMBparams` instance with cosmological parameters etc.
- **CMB_unit** – scale results from dimensionless, use ‘muK’ for μK^2 units

Returns dictionary of power spectra, with $L(L + 1)/2\pi$ factors.

Matter power spectrum and matter transfer function variables

The various matter power spectrum functions, e.g. `get_matter_power_interpolator()`, can calculate power spectra for various quantities. Each variable used to form the power spectrum has a name as follows:

name	number	description
k/h	1	k/h
delta_cdm	2	Δ_c , CDM density
delta_baryon	3	Δ_b , baryon density
delta_photon	4	Δ_γ , photon density
delta_neutrino	5	Δ_ν , for massless neutrinos
delta_nu	6	Δ_ν for massive neutrinos
delta_tot	7	$\frac{\rho_c \Delta_c + \rho_b \Delta_b + \rho_\nu \Delta_\nu}{\rho_c + \rho_b + \rho_\nu}$, CDM+baryons+massive neutrino density
delta_nonu	8	$\frac{\rho_c \Delta_c + \rho_b \Delta_b}{\rho_b + \rho_c}$, CDM+baryon density
delta_tot_de	9	$\frac{\rho_c \Delta_c + \rho_b \Delta_b + \rho_\nu \Delta_\nu + \rho_{de} \Delta_{de}}{\rho_c + \rho_b + \rho_\nu}$, CDM+baryons+massive neutrinos+ dark energy (numerator only) density
Weyl	10	$k^2 \Psi \equiv k^2 (\phi + \psi)/2$, the Weyl potential scaled by k^2 to scale in k like a density.
v_newtonian_cdm	11	$-v_{N,c} k/\mathcal{H}$ (where $v_{N,c}$ is the Newtonian-gauge CDM velocity)
v_newtonian_baryon	12	$-v_{N,b} k/\mathcal{H}$ (Newtonian-gauge baryon velocity $v_{N,b}$)
v_baryon_cdm	13	$v_b - v_c$, relative baryon-CDM velocity

The number here corresponds to a corresponding numerical index, in Fortran these are the same as *model.name*, where *name* are the Transfer_xxx variable names: Transfer_kh=1, Transfer_cdm=2, Transfer_b=3, Transfer_g=4, Transfer_r=5, Transfer_nu=6, Transfer_tot=7, Transfer_nonu=8, Transfer_tot_de=9, Transfer_Weyl=10, Transfer_Newt_vel_cdm=11, Transfer_Newt_vel_baryon=12, Transfer_vel_baryon_cdm = 13.

So for example, requesting `var1='delta_b'`, `var2='Weyl'` or alternatively `var1=model.Transfer_b`, `var2=model.Transfer_Weyl` would get the power spectrum for the cross-correlation of the baryon density with the Weyl potential. All density variables Δ_i here are synchronous gauge.

For transfer function variables (rather than matter power spectra), the variables are normalized corresponding to unit primordial curvature perturbation on super-horizon scales. The `get_matter_transfer_data()` function returns the above quantities divided by k^2 (so they are roughly constant at low k on super-horizon scales).

The [example notebook](#) has various examples of getting the matter power spectrum, relating the Weyl-potential spectrum to lensing, and calculating the baryon-dark matter relative velocity spectra. There is also an explicit example of how to calculate the matter power spectrum manually from the matter transfer functions.

When generating dark-age 21cm power spectra (do21cm is set) the transfer functions are instead the *model.name* variables (see equations 20 and 25 of [astro-ph/0702600](#))

name	number	description
Transfer_kh	1	k/h
Transfer_cdm	2	Δ_c , CDM density
Transfer_b	3	Δ_b , baryon density
Transfer_monopole	4	$\Delta_s + (r_\tau - 1)(\Delta_b - \Delta_{T_s})$, 21cm monopole source
Transfer_vnewt	5	$r_\tau k v_{N,b} / \mathcal{H}$, 21cm Newtonian-gauge velocity source
Transfer_Tmat	6	Δ_{T_m} , matter temperature perturbation
Transfer_tot	7	$\frac{\rho_c \Delta_c + \rho_b \Delta_b + \rho_\nu \Delta_\nu}{\rho_c + \rho_b + \rho_\nu}$, CDM+baryons+massive neutrino density
Transfer_nonu	8	$\frac{\rho_c \Delta_c + \rho_b \Delta_b}{\rho_b + \rho_c}$, CDM+baryon density
Transfer_tot_de	9	$\frac{\rho_c \Delta_c + \rho_b \Delta_b + \rho_\nu \Delta_\nu + \rho_{de} \Delta_{de}}{\rho_c + \rho_b + \rho_\nu}$, CDM+baryons+massive neutrinos+ dark energy (numerator only) density
Transfer_Weyl	10	$k^2 \Psi \equiv k^2 (\phi + \psi) / 2$, the Weyl potential scaled by k^2 to scale in k like a density.
Transfer_Newt_vel_cdm	11	$-v_{N,c} k / \mathcal{H}$ (where $v_{N,c}$ is the Newtonian-gauge CDM velocity)
Transfer_Newt_vel_baryon	12	$-v_{N,b} k / \mathcal{H}$ (Newtonian-gauge baryon velocity $v_{N,b}$)
Transfer_vel_baryon_cdm	13	$v_b - v_c$, relative baryon-CDM velocity

If use_21cm_mK is set the 21cm results are multiplied by T_b to give results in mK units.

Fortran compilers

CAMB internally uses modern (object-oriented) Fortran 2008 for most numerical calculations, and needs a recent fortran compiler to build the numerical library. The recommended compilers are

- gfortran version 6.3 or higher
- Intel Fortran (ifort), version 18.0.1 or higher (some things may work with version 14+)

The gfortran compiler is part of the standard “gcc” compiler package, and may be pre-installed on recent unix systems. Check the version using “gfortran –version”.

If you do not have a suitable Fortran compiler, you can get one as follows:

Mac Download the [binary installation](#)

Windows Download gfortran as part of [MinGW-w64](#) (select x86_64 option in the installation program)

Linux To install from the standard repository use:

- “sudo apt-get update; sudo apt-get install gfortran”

On Ubuntu systems where the default gfortran is too old, you can use this to install a later version

- sudo add-apt-repository ppa:ubuntu-toolchain-r/test
- sudo apt-get update
- sudo apt install gfortran-8

To make this the default gfortran then use

- mkdir -p gfortran-symlinks
- ln -s /usr/bin/gfortran-8 gfortran-symlinks/gfortran
- export PATH=\$PWD/gfortran-symlinks:\$PATH

To re-use next time, add gfortran-symlinks directory to your startup settings (.bashrc).

Alternatively you can compile and run in a container or virtual machine: e.g., see [CosmoBox](#). For example, to run a configured shell in docker where you can install and run camb from the command line (after changing to the camb directory):

```
docker run -v /local/git/path/CAMB:/camb -i -t cmbant/cosmobox
```

Updating and modified Fortran code

In the main CAMB source root directory, to re-build the Fortran binary including any pulled or local changes use:

```
python setup.py make
```

This will also work on Windows as long as you have MinGW-w64 installed as described above.

Note that you will need to close all python instances using camb before you can re-load with an updated library. This includes in Jupyter notebooks; just re-start the kernel or use:

```
import IPython
IPython.Application.instance().kernel.do_shutdown(True)
```

If you want to automatically rebuild the library from Jupyter you can do something like this:

```
import subprocess
import sys
import os
src_dir = '/path/to/git/CAMB'
try:
    subprocess.check_output(r'python "%s" make'%os.path.join(src_dir, 'setup.py'),
                            stderr=subprocess.STDOUT)
    sys.path.insert(0,src_dir)
    import camb
    print('Using CAMB %s installed at %s'%(camb.__version__,
                                           os.path.dirname(camb.__file__)))
except subprocess.CalledProcessError as E:
    print(E.output.decode())
```


This module contains some fast utility functions that are useful in the same contexts as `camb`. They are entirely independent of the main `camb` code.

`camb.mathutils.chi_squared(covinv, x)`

Utility function to efficiently calculate $x^T \text{covinv} x$

Parameters

- **covinv** – symmetric inverse covariance matrix
- **x** – vector

Returns `covinv.dot(x).dot(x)`, but parallelized and using symmetry

`camb.mathutils.pcl_coupling_matrix(P, lmax, pol=False)`

Get Pseudo-Cl coupling matrix from power spectrum of mask. Uses multiple threads. See Eq A31 of [astro-ph/0105302](#)

Parameters

- **P** – power spectrum of mask
- **lmax** – `lmax` for the matrix
- **pol** – whether to calculate TE, EE, BB couplings

Returns coupling matrix (square but not symmetric), or list of TT, TE, EE, BB if `pol`

`camb.mathutils.scalar_coupling_matrix(P, lmax)`

Get scalar Pseudo-Cl coupling matrix from power spectrum of mask, or array of power masks. Uses multiple threads. See Eq A31 of [astro-ph/0105302](#)

Parameters

- **P** – power spectrum of mask, or list of mask power spectra
- **lmax** – `lmax` for the matrix (assumed square)

Returns coupling matrix (square but not symmetric), or list of couplings for different masks

`camb.mathutils.threej(l2, l3, m2, m3)`

Convenience wrapper around standard 3j function, returning array for all allowed l1 values

Parameters

- **l2** – L_2
- **l3** – L_3
- **m2** – M_2
- **m3** – M_3

Returns array of 3j from $\max(\text{abs}(l2-l3), \text{abs}(m2+m3)) .. l2+l3$

`camb.mathutils.threej_coupling(W, lmax, pol=False)`

Calculate symmetric coupling matrix $\langle \tilde{C}_\ell \rangle$ for given weights W_ℓ , where $\langle \tilde{C}_\ell \rangle = \Xi_{\ell\ell'}(2\ell' + 1)C_\ell$. The weights are related to the power spectrum of the mask P by $W_\ell = (2\ell + 1)P_\ell/4\pi$. See e.g. Eq D16 of [arxiv:0801.0554](https://arxiv.org/abs/0801.0554).

If pol is False and W is an array of weights, produces array of temperature couplings, otherwise for pol is True produces set of TT, TE, EE, EB couplings (and weights must have one spectrum - for same masks - or three).

Use `scalar_coupling_matrix()` or `pcl_coupling_matrix()` to get the coupling matrix directly from the mask power spectrum.

Parameters

- **W** – 1d array of Weights for each L, or list of arrays of weights (zero based)
- **lmax** – lmax for the output matrix (assumed symmetric, though not in principle)
- **pol** – if pol, produce TT, TE, EE, EB couplings for three input mask weights (or one if assuming same mask)

Returns symmetric coupling matrix or array of matrices

- [Example notebook](#)
- [genindex](#)

C

`camb`, 3
`camb.bbn`, 35
`camb.correlations`, 53
`camb.emission_angle`, 61
`camb.mathutils`, 69
`camb.postborn`, 59
`camb.symbolic`, 31

A

AccuracyParams (class in *camb.model*), 14
 angular_diameter_distance() (camb.results.CAMBdata method), 18
 angular_diameter_distance2() (camb.results.CAMBdata method), 18
 AxionEffectiveFluid (class in *camb.dark_energy*), 40

B

BBN_fitting_parthenope (class in *camb.bbn*), 35
 BBN_table_interpolator (class in *camb.bbn*), 36
 BBNInterpolator (class in *camb.bbn*), 35
 BBNPredictor (class in *camb.bbn*), 35

C

calc_background() (camb.results.CAMBdata method), 18
 calc_background_no_thermo() (camb.results.CAMBdata method), 19
 calc_power_spectra() (camb.results.CAMBdata method), 19
 calc_transfers() (camb.results.CAMBdata method), 19
 camb (module), 3
 camb.bbn (module), 35
 camb.correlations (module), 53
 camb.emission_angle (module), 61
 camb.mathutils (module), 69
 camb.postborn (module), 59
 camb.symbolic (module), 31
 camb_fortran() (in module *camb.symbolic*), 31
 CAMBdata (class in *camb.results*), 17
 CAMBparams (class in *camb.model*), 7
 cdm_gauge() (in module *camb.symbolic*), 32
 chi_squared() (in module *camb.mathutils*), 69
 cl2corr() (in module *camb.correlations*), 53
 ClTransferData (class in *camb.results*), 30

comoving_radial_distance() (camb.results.CAMBdata method), 19
 compile_source_function_code() (in module *camb.symbolic*), 32
 conformal_time() (camb.results.CAMBdata method), 19
 conformal_time_a1_a2() (camb.results.CAMBdata method), 19
 copy() (camb.model.CAMBparams method), 9
 copy() (camb.results.CAMBdata method), 20
 corr2cl() (in module *camb.correlations*), 53
 cosmomc_theta() (camb.results.CAMBdata method), 20
 CosmoRec (class in *camb.recombination*), 49
 CustomSources (class in *camb.model*), 16

D

DarkEnergyEqnOfState (class in *camb.dark_energy*), 39
 DarkEnergyFluid (class in *camb.dark_energy*), 40
 DarkEnergyModel (class in *camb.dark_energy*), 39
 DarkEnergyPPF (class in *camb.dark_energy*), 40
 DH() (camb.bbn.BBN_table_interpolator method), 36
 diff() (camb.model.CAMBparams method), 9

F

f_K (class in *camb.symbolic*), 32

G

gauss_legendre_correlation() (in module *camb.correlations*), 54
 GaussianSourceWindow (class in *camb.sources*), 51
 get() (camb.bbn.BBN_table_interpolator method), 36
 get_age() (in module *camb*), 6
 get_background() (in module *camb*), 3
 get_background_densities() (camb.results.CAMBdata method), 20
 get_background_outputs() (camb.results.CAMBdata method), 20

`get_background_redshift_evolution()` (*camb.results.CAMBdata method*), 20
`get_background_time_evolution()` (*camb.results.CAMBdata method*), 21
`get_BAO()` (*camb.results.CAMBdata method*), 20
`get_cmb_correlation_functions()` (*camb.results.CAMBdata method*), 21
`get_cmb_power_spectra()` (*camb.results.CAMBdata method*), 21
`get_cmb_transfer_data()` (*camb.results.CAMBdata method*), 22
`get_cmb_unlensed_scalar_array_dict()` (*camb.results.CAMBdata method*), 22
`get_dark_energy_rho_w()` (*camb.results.CAMBdata method*), 22
`get_derived_params()` (*camb.results.CAMBdata method*), 22
`get_DH()` (*camb.model.CAMBparams method*), 9
`get_emission_angle_powers()` (*in module camb.emission_angle*), 61
`get_emission_delay_BB()` (*in module camb.emission_angle*), 61
`get_field_rotation_BB()` (*in module camb.postborn*), 59
`get_field_rotation_power()` (*in module camb.postborn*), 59
`get_fsigma8()` (*camb.results.CAMBdata method*), 22
`get_hierarchies()` (*in module camb.symbolic*), 32
`get_lens_potential_cls()` (*camb.results.CAMBdata method*), 22
`get_lensed_gradient_cls()` (*camb.results.CAMBdata method*), 23
`get_lensed_scalar_cls()` (*camb.results.CAMBdata method*), 23
`get_linear_matter_power_spectrum()` (*camb.results.CAMBdata method*), 23
`get_matter_power_interpolator()` (*camb.results.CAMBdata method*), 23
`get_matter_power_interpolator()` (*in module camb*), 5
`get_matter_power_spectrum()` (*camb.results.CAMBdata method*), 24
`get_matter_transfer_data()` (*camb.results.CAMBdata method*), 24
`get_nonlinear_matter_power_spectrum()` (*camb.results.CAMBdata method*), 25
`get_Omega()` (*camb.results.CAMBdata method*), 20
`get_predictor()` (*in module camb.bbn*), 36
`get_redshift_evolution()` (*camb.results.CAMBdata method*), 25
`get_results()` (*in module camb*), 3
`get_scalar_temperature_sources()` (*in module camb.symbolic*), 32
`get_sigma8()` (*camb.results.CAMBdata method*), 25
`get_sigma8_0()` (*camb.results.CAMBdata method*), 25
`get_sigmaR()` (*camb.results.CAMBdata method*), 25
`get_source_cls_dict()` (*camb.results.CAMBdata method*), 26
`get_source_cmb_cl()` (*in module camb.emission_angle*), 62
`get_tensor_cls()` (*camb.results.CAMBdata method*), 26
`get_time_evolution()` (*camb.results.CAMBdata method*), 26
`get_total_cls()` (*camb.results.CAMBdata method*), 27
`get_transfer()` (*camb.results.CITransferData method*), 30
`get_transfer_functions()` (*in module camb*), 3
`get_unlensed_scalar_array_cls()` (*camb.results.CAMBdata method*), 27
`get_unlensed_scalar_cls()` (*camb.results.CAMBdata method*), 27
`get_unlensed_total_cls()` (*camb.results.CAMBdata method*), 27
`get_valid_numerical_params()` (*in module camb*), 6
`get_Y_p()` (*camb.model.CAMBparams method*), 9
`get_zre()` (*camb.reionization.TanhReionization method*), 48
`get_zre_from_tau()` (*in module camb*), 6

H

`h_of_z()` (*camb.results.CAMBdata method*), 27
Halofit (*class in camb.nonlinear*), 45
`has_tensors()` (*camb.initialpower.InitialPowerLaw method*), 41
`has_tensors()` (*camb.initialpower.SplinedInitialPower method*), 42
`hubble_parameter()` (*camb.results.CAMBdata method*), 27
HyRec (*class in camb.recombination*), 50

I

InitialPower (*class in camb.initialpower*), 41
InitialPowerLaw (*class in camb.initialpower*), 41

L

`legendre_funcs()` (*in module camb.correlations*), 54
`lensed_cl_derivative_unlensed()` (*in module camb.correlations*), 54
`lensed_cl_derivatives()` (*in module camb.correlations*), 55
`lensed_cls()` (*in module camb.correlations*), 55

lensed_correlations() (in module *camb.correlations*), 56
 lensing_correlations() (in module *camb.correlations*), 56
 lensing_R() (in module *camb.correlations*), 56
 LinearPerturbation() (in module *camb.symbolic*), 31
 luminosity_distance() (*camb.results.CAMBdata* method), 28

M

make_frame_invariant() (in module *camb.symbolic*), 32
 MatterTransferData (class in *camb.results*), 29

N

N_eff (*camb.model.CAMBparams* attribute), 9
 newtonian_gauge() (in module *camb.symbolic*), 32
 NonLinearModel (class in *camb.nonlinear*), 45

P

pcl_coupling_matrix() (in module *camb.mathutils*), 69
 physical_time() (*camb.results.CAMBdata* method), 28
 physical_time_a1_a2() (*camb.results.CAMBdata* method), 28
 power_spectra_from_transfer() (*camb.results.CAMBdata* method), 28

R

read_ini() (in module *camb*), 5
 Recfast (class in *camb.recombination*), 49
 RecombinationModel (class in *camb.recombination*), 49
 redshift_at_comoving_radial_distance() (*camb.results.CAMBdata* method), 28
 redshift_at_conformal_time() (*camb.results.CAMBdata* method), 28
 ReionizationModel (class in *camb.reionization*), 47
 replace() (*camb.model.CAMBparams* method), 10
 replace() (*camb.results.CAMBdata* method), 29
 run_ini() (in module *camb*), 6

S

save_cmb_power_spectra() (*camb.results.CAMBdata* method), 29
 scalar_coupling_matrix() (in module *camb.mathutils*), 69
 scalar_power() (*camb.model.CAMBparams* method), 10
 SecondOrderPK (class in *camb.nonlinear*), 46
 set_accuracy() (*camb.model.CAMBparams* method), 10
 set_classes() (*camb.model.CAMBparams* method), 10
 set_cosmology() (*camb.model.CAMBparams* method), 11
 set_custom_scalar_sources() (*camb.model.CAMBparams* method), 12
 set_dark_energy() (*camb.model.CAMBparams* method), 12
 set_dark_energy_w_a() (*camb.model.CAMBparams* method), 12
 set_feedback_level() (in module *camb*), 6
 set_for_lmax() (*camb.model.CAMBparams* method), 12
 set_H0_for_theta() (*camb.model.CAMBparams* method), 10
 set_initial_power() (*camb.model.CAMBparams* method), 13
 set_initial_power_function() (*camb.model.CAMBparams* method), 13
 set_initial_power_table() (*camb.model.CAMBparams* method), 13
 set_matter_power() (*camb.model.CAMBparams* method), 14
 set_nonlinear_lensing() (*camb.model.CAMBparams* method), 14
 set_params() (*camb.dark_energy.DarkEnergyEqnOfState* method), 39
 set_params() (*camb.initialpower.InitialPowerLaw* method), 41
 set_params() (*camb.nonlinear.Halofit* method), 45
 set_params() (*camb.results.CAMBdata* method), 29
 set_params() (in module *camb*), 3
 set_params_cosmomc() (in module *camb*), 4
 set_scalar_log_regular() (*camb.initialpower.SplinedInitialPower* method), 42
 set_scalar_table() (*camb.initialpower.SplinedInitialPower* method), 42
 set_table() (*camb.sources.SplinedSourceWindow* method), 51
 set_tau() (*camb.reionization.TanhReionization* method), 48
 set_tensor_log_regular() (*camb.initialpower.SplinedInitialPower* method), 42
 set_tensor_table() (*camb.initialpower.SplinedInitialPower* method), 43
 set_w_a_table() (*camb.dark_energy.DarkEnergyEqnOfState* method), 39
 set_zrei() (*camb.reionization.TanhReionization* method), 48
 sound_horizon() (*camb.results.CAMBdata*

method), 29

SourceTermParams (*class in camb.model*), 15

SourceWindow (*class in camb.sources*), 51

SplinedInitialPower (*class in camb.initialpower*),
42

SplinedSourceWindow (*class in camb.sources*), 51

synchronous_gauge() (*in module camb.symbolic*),
33

T

TanhReionization (*class in camb.reionization*), 47

tensor_power() (*camb.model.CAMBparams
method*), 14

threej() (*in module camb.mathutils*), 69

threej_coupling() (*in module camb.mathutils*), 70

transfer_z() (*camb.results.MatterTransferData
method*), 30

TransferParams (*class in camb.model*), 15

V

validate() (*camb.model.CAMBparams method*), 14

Y

Y_He() (*camb.bbn.BBNPredictor method*), 35

Y_p() (*camb.bbn.BBN_fitting_parthenope method*), 35

Y_p() (*camb.bbn.BBN_table_interpolator method*), 36

Y_p() (*camb.bbn.BBNPredictor method*), 35