pymaster Documentation

Release 0.1

David Alonso

Dec 20, 2019

Contents:

pymaster is the python implementation of the NaMaster library. The main purpose of this library is to provide support to compute the angular power spectrum of fields defined on a limited region of the sphere using the so-called pseudo-CL formalism.

Below you can find links to pymaster's full documentation and 8 different example scripts showcasing its usage. Understanding the last script in particular will allow you to make the most efficient use of this module.

We recommend that users read NaMaster's `scientific documentation` and the C API documentation for a complete overview of all the facilities included in this package, and in order to have a good understanding of the methods implemented in it.

CHAPTER 1

Python API documentation

pymaster contains three basic classes:

- *pymaster.field.NmtField*
- *pymaster.bins.NmtBin*
- *pymaster.workspaces.NmtWorkspace*
- *pymaster.covariance.NmtCovarianceWorkspace*

and a number of functions

- *pymaster.workspaces.deprojection_bias()*
- *pymaster.workspaces.compute_coupled_cell()*
- *pymaster.workspaces.compute_full_master()*
- *pymaster.covariance.gaussian_covariance()*
- *pymaster.utils.mask_apodization()*
- *pymaster.utils.synfast_spherical()*

pymaster also comes with a flat-sky version with most of the same functionality:

- *pymaster.field.NmtFieldFlat*
- *pymaster.bins.NmtBinFlat*
- *pymaster.workspaces.NmtWorkspaceFlat*
- *pymaster.workspaces.deprojection_bias_flat()*
- *pymaster.workspaces.compute_coupled_cell_flat()*
- *pymaster.workspaces.compute_full_master_flat()*
- *pymaster.covariance.gaussian_covariance_flat()*
- *pymaster.utils.mask_apodization_flat()*
- *pymaster.utils.synfast_flat()*

Many of these function accept or return sets of power spectra (arrays with one element per angular multipole) or bandpowers (binned versions of power spectra). In all cases, these are returned and provided as 2D arrays with shape [n_cls][nl], where n_cls is the number of power spectra and nl is either the number of multipoles or bandpowers. In all cases, n_cls should correspond with the spins of the two fields being correlated, and the ordering is as follows:

- Two spin-0 fields: n_cls=1, [C_T1T2]
- One spin-0 field and one spin-2 field: n_cls=2, [C_TE,C_TB]
- Two spin-2 fields: n_cls=4, [C_E1E2,C_E1B2,C_E2B1,C_B1B2]

All sky maps accepted and returned by these functions are in the form of HEALPix maps exclusively with RING ordering.

class pymaster.field.**NmtField**(*mask*, *maps*, *templates=None*, *beam=None*, *purify_e=False*, *purify_b=False*, *n_iter_mask_purify=3*, *tol_pinv=1e-10*, *wcs=None*, *n_iter=3*, *lmax_sht=-1*)

An NmtField object contains all the information describing the fields to correlate, including their observed maps, masks and contaminant templates.

Parameters

- **mask** – array containing a map corresponding to the field's mask. Should be 1-dimensional for a HEALPix map or 2-dimensional for a map with rectangular pixelization.
- **maps** – array containing the observed maps for this field. Should be at least 2-dimensional. The first dimension corresponds to the number of maps, which should be 1 for a spin-0 field and 2 for a spin-2 field. The other dimensions should be [npix] for HEALPix maps or [ny,nx] for maps with rectangular pixels. For a spin-2 field, the two maps to pass should be the usual Q/U Stokes parameters for polarization, or e1/e2 (gamma1/gamma2 etc.) in the case of cosmic shear. It is important to note that NaMaster uses the same polarization convention as HEALPix (i.e. with the x-coordinate growing with increasing colatitude theta). It is however more common for galaxy ellipticities to be provided using the IAU convention (i.e. x grows with declination). In this case, the sign of the e2/gamma2 map should be swapped before using it to create an NmtField. See more here .
- **templates** – array containing a set of contaminant templates for this field. This array should have shape [ntemp][nmap]..., where ntemp is the number of templates, nmap should be 1 for spin-0 fields and 2 for spin-2 fields. The other dimensions should be [npix] for HEALPix maps or [ny,nx] for maps with rectangular pixels. The best-fit contribution from each contaminant is automatically removed from the maps unless templates=None.
- **beam** – spherical harmonic transform of the instrumental beam (assumed to be rotationally symmetric - i.e. no m dependence). If None, no beam will be corrected for. Otherwise, this array should have at least as many elements as the maximum multipole sampled by the maps + 1 (e.g. if a HEALPix map, it should contain 3*nside elements, corresponding to multipoles from 0 to 3*nside-1).
- **purify_e** – use pure E-modes?
- **purify_b** – use pure B-modes?
- **n_iter_mask_purify** – number of iterations used to compute an accurate SHT of the mask when using E/B purification
- **tol_pinv** – when computing the pseudo-inverse of the contaminant covariance matrix, all eigenvalues below tol_pinv * max_eval will be treated as singular values, where max_eval is the largest eigenvalue. Only relevant if passing contaminant templates that are likely to be highly correlated.
- **wcs** – a WCS object if using rectangular pixels (see http://docs.astropy.org/en/stable/wcs/index.html).

- **n_iter** – number of iterations when computing a_lms.
- **lmax_sht** – maximum multipole up to which map power spectra will be computed. If negative or zero, the maximum multipole given the map resolution will be used (e.g. 3 * nside - 1 for HEALPix maps).

get_maps()
Returns a 2D array ([nmap][npix]) corresponding to the observed maps for this field. If the field was initialized with contaminant templates, the maps returned by this function have their best-fit contribution from these contaminants removed.

Returns 2D array of maps

get_templates()
Returns a 3D array ([ntemp][nmap][npix]) corresponding to the contaminant templates passed when initializing this field.

Returns 3D array of maps

class pymaster.field.**NmtFieldFlat**(*lx*, *ly*, *mask*, *maps*, *templates=None*, *beam=None*, *purify_e=False*, *purify_b=False*, *tol_pinv=1e-10*)
An NmtFieldFlat object contains all the information describing the flat-sky fields to correlate, including their observed maps, masks and contaminant templates.

Parameters

- **lx,ly** (*float*) – size of the patch in the x and y directions (in radians)
- **mask** – 2D array (nx,ny) containing a HEALPix map corresponding to the field's mask.
- **maps** – 2 2D arrays (nmaps,nx,ny) containing the observed maps for this field. The first dimension corresponds to the number of maps, which should be 1 for a spin-0 field and 2 for a spin-2 field.
- **templates** – array of maps (ntemp,nmaps,nx,ny) containing a set of contaminant templates for this field. This array should have shape [ntemp][nmap][nx][ny], where ntemp is the number of templates, nmap should be 1 for spin-0 fields and 2 for spin-2 fields, and nx,ny define the patch. The best-fit contribution from each contaminant is automatically removed from the maps unless templates=None
- **beam** – 2D array (2,nl) defining the FT of the instrumental beam (assumed to be rotationally symmetric). beam[0] should contain the values of l for which de beam is defined, with beam[1] containing the beam values. If None, no beam will be corrected for.
- **purify_e** – use pure E-modes?
- **purify_b** – use pure B-modes?
- **tol_pinv** – when computing the pseudo-inverse of the contaminant covariance matrix, all eigenvalues below tol_pinv * max_eval will be treated as singular values, where max_eval is the largest eigenvalue. Only relevant if passing contaminant templates that are likely to be highly correlated.

get_maps()
Returns a 3D array ([nmap][ny][nx]) corresponding to the observed maps for this field. If the field was initialized with contaminant templates, the maps returned by this function have their best-fit contribution from these contaminants removed.

Returns 3D array of flat-sky maps

get_templates()
Returns a 4D array ([ntemp][nmap][ny][nx]) corresponding to the contaminant templates passed when initializing this field.

Returns 4D array of flat-sky maps

class pymaster.bins.**NmtBin**(*nside=None*, *bpws=None*, *ells=None*, *weights=None*, *nlb=None*, *lmax=None*, *is_Dell=False*, *f_ell=None*)

An NmtBin object defines the set of bandpowers used in the computation of the pseudo-Cl estimator. The definition of bandpowers is described in Section 3.6 of the scientific documentation. We provide several convenience constructors that cover a range of common use cases requiring fewer parameters (see *NmtBin.from_nside_linear()*, *NmtBin.from_lmax_linear()* and `Nmt.from_edges()`).

Parameters

- **nside** (*int*) – HEALPix nside resolution parameter of the maps you intend to correlate. The maximum multipole considered for bandpowers will be 3*nside-1, unless *lmax* is set.
- **ells** (*array-like*) – array of integers corresponding to different multipoles
- **bpws** (*array-like*) – array of integers that assign the multipoles in ells to different bandpowers. All negative values will be ignored.
- **weights** (*array-like*) – array of floats corresponding to the weights associated to each multipole in ells. The sum of weights within each bandpower is normalized to 1.
- **nlb** (*int*) – integer value corresponding to a constant bandpower width. I.e. the bandpowers will be defined as consecutive sets of nlb multipoles from l=2 to l=lmax (see below) with equal weights. If this argument is provided, the values of ells, bpws and weights are ignored.
- **lmax** (*int*) – integer value corresponding to the maximum multipole used by these bandpowers. If None, it will be set to 3*nside-1. In any case the actual maximum multipole will be chosen as the minimum of lmax, 3*nside-1 and the maximum element of ells (e.g. if you are using CAR maps and don't care about nside, you can pass whatever lmax you want and e.g. nside=lmax).
- **is_Dell** (*boolean*) – if True, the output of all pseudo-Cl computations carried out using this bandpower scheme (e.g. from *pymaster.workspaces.NmtWorkspace.decouple_cell()*) will be multiplied by *ell * (ell + 1) / 2 * PI*, where *ell* is the multipole order (no prefactor otherwise).
- **f_ell** (*array-like*) – if present, this is array represents an *ell-dependent* function that will be multiplied by all pseudo-Cl computations carried out using this bandpower scheme. If not *None*, the value of *is_Dell* is ignored.

bin_cell(*cls_in*)

Bins a power spectrum into bandpowers. This is carried out as a weighted average over the multipoles in each bandpower.

Parameters **cls_in** (*array-like*) – 2D array of power spectra

Returns array of bandpowers

classmethod from_edges(*ell_ini*, *ell_end*, *is_Dell=False*)

Convenience constructor for general equal-weight bands. All ells in the interval [ell_ini, ell_end) will be binned with equal weights across the band.

Parameters

- **ell_ini** (*int*) – array containing the lower edges of each bandpower.
- **ell_end** (*int*) – array containing the upper edges of each bandpower.
- **is_Dell** (*boolean*) – if True, the output of all pseudo-Cl computations carried out using this bandpower scheme (e.g. from *pymaster.workspaces.NmtWorkspace.*

decouple_cell()) will be multiplied by *ell * (ell + 1) / 2 * PI*, where *ell* is the multipole order (no prefactor otherwise).

classmethod from_lmax_linear(*lmax*, *nlb*, *is_Dell=False*)
Convenience constructor for generic linear binning.

Parameters

- **lmax** (*int*) – integer value corresponding to the maximum multipole used by these bandpowers.
- **nlb** (*int*) – integer value corresponding to a constant bandpower width. I.e. the bandpowers will be defined as consecutive sets of nlb multipoles from l=2 to l=lmax with equal weights.
- **is_Dell** (*boolean*) – if True, the output of all pseudo-Cl computations carried out using this bandpower scheme (e.g. from pymaster.workspaces.NmtWorkspace.decouple_cell()) will be multiplied by *ell * (ell + 1) / 2 * PI*, where *ell* is the multipole order (no prefactor otherwise).

classmethod from_nside_linear(*nside*, *nlb*, *is_Dell=False*)
Convenience constructor for HEALPix maps with linear binning.

Parameters

- **nside** (*int*) – HEALPix nside resolution parameter of the maps you intend to correlate. The maximum multipole considered for bandpowers will be 3*nside-1.
- **nlb** (*int*) – integer value corresponding to a constant bandpower width. I.e. the bandpowers will be defined as consecutive sets of nlb multipoles from l=2 to l=lmax with equal weights.
- **is_Dell** (*boolean*) – if True, the output of all pseudo-Cl computations carried out using this bandpower scheme (e.g. from pymaster.workspaces.NmtWorkspace.decouple_cell()) will be multiplied by *ell * (ell + 1) / 2 * PI*, where *ell* is the multipole order (no prefactor otherwise).

get_effective_ells()
Returns an array with the effective multipole associated to each bandpower. These are computed as a weighted average of the multipoles within each bandpower.

Returns effective multipoles for each bandpower

get_ell_list(*ibin*)
Returns an array with the multipoles in the ibin-th bandpower

Parameters **ibin** (*int*) – bandpower index

Returns multipoles associated with bandpower ibin

get_n_bands()
Returns the number of bandpowers stored in this object

Returns number of bandpowers

get_nell_list()
Returns an array with the number of multipoles in each bandpower stored in this object

Returns number of multipoles per bandpower

get_weight_list(*ibin*)
Returns an array with the weights associated to each multipole in the ibin-th bandpower

Parameters **ibin** (*int*) – bandpower index

Returns weights associated to multipoles in bandpower ibin

unbin_cell(*cls_in*)
Un-bins a set of bandpowers into a power spectrum. This is simply done by assigning a constant value for every multipole in each bandpower (corresponding to the value of that bandpower).

Parameters **cls_in** (*array-like*) – array of bandpowers

Returns array of power spectra

class pymaster.bins.**NmtBinFlat**(*l0*, *lf*)
An NmtBinFlat object defines the set of bandpowers used in the computation of the pseudo-Cl estimator. The definition of bandpowers is described in Section 3.6 of the scientific documentation. Note that currently pymaster only supports top-hat bandpowers for flat-sky power spectra.

Parameters

- **l0** (*array-like*) – array of floats corresponding to the lower bound of each bandpower.
- **lf** (*array-like*) – array of floats corresponding to the upper bound of each bandpower. lf should have the same shape as l0

bin_cell(*ells*, *cls_in*)
Bins a power spectrum into bandpowers. This is carried out as a weighted average over the multipoles in each bandpower.

Parameters

- **ells** (*array-like*) – multipole values at which the input power spectra are defined
- **cls_in** (*array-like*) – 2D array of input power spectra

Returns array of bandpowers

get_effective_ells()
Returns an array with the effective multipole associated to each bandpower. These are computed as a weighted average of the multipoles within each bandpower.

Returns effective multipoles for each bandpower

get_n_bands()
Returns the number of bandpowers stored in this object

Returns number of bandpowers

unbin_cell(*cls_in*, *ells*)
Un-bins a set of bandpowers into a power spectrum. This is simply done by assigning a constant value for every multipole in each bandpower (corresponding to the value of that bandpower).

Parameters

- **cls_in** (*array-like*) – array of bandpowers
- **ells** (*array-like*) – array of multipoles at which the power spectra should be intepolated

Returns array of power spectra

class pymaster.workspaces.**NmtWorkspace**
NmtWorkspace objects are used to compute and store the coupling matrix associated with an incomplete sky coverage, and used in the MASTER algorithm. When initialized, this object is practically empty. The information describing the coupling matrix must be computed or read from a file afterwards.

compute_coupling_matrix(*fl1*, *fl2*, *bins*, *is_teb=False*, *n_iter=3*, *lmax_mask=-1*)
Computes coupling matrix associated with the cross-power spectrum of two NmtFields and an NmtBin

binning scheme. Note that the mode coupling matrix will only contain ells up to the maximum multipole included in the NmtBin bandpowers.

Parameters

- **fl1,fl2** (NmtField) – fields to correlate
- **bin** (NmtBin) – binning scheme
- **is_teb** (*boolean*) – if true, all mode-coupling matrices (0-0,0-2,2-2) will be computed at the same time. In this case, fl1 must be a spin-0 field and fl1 must be spin-2.
- **n_iter** – number of iterations when computing a_lms.
- **lmax_mask** – maximum multipole for masks. If smaller than the maximum multipoles of the fields, it will be set to that.

couple_cell(*cl_in*)

Convolves a set of input power spectra with a coupling matrix (see Eq. 6 of the C API documentation).

Parameters **cl_in** – set of input power spectra. The number of power spectra must correspond to the spins of the two fields that this NmtWorkspace object was initialized with (i.e. 1 for two spin-0 fields, 2 for one spin-0 and one spin-2 field and 4 for two spin-2 fields).

Returns coupled power spectrum

decouple_cell(*cl_in*, *cl_bias=None*, *cl_noise=None*)

Decouples a set of pseudo-Cl power spectra into a set of bandpowers by inverting the binned coupling matrix (se Eq. 4 of the C API documentation).

Parameters

- **cl_in** – set of input power spectra. The number of power spectra must correspond to the spins of the two fields that this NmtWorkspace object was initialized with (i.e. 1 for two spin-0 fields, 2 for one spin-0 and one spin-2 field, 4 for two spin-2 fields and 7 if this NmtWorkspace was created using *is_teb=True*).
- **cl_bias** – bias to the power spectrum associated to contaminant residuals (optional). This can be computed through `pymaster.deprojection_bias()`.
- **cl_noise** – noise bias (i.e. angular power spectrum of masked noise realizations).

Returns set of decoupled bandpowers

get_bandpower_windows()

Get bandpower window functions. Convolve the theory power spectra with these as an alternative to the combination *decouple_cell(couple_cell(*.

Returns bandpower windows with shape *[n_cls, n_bpws, n_cls, lmax+1]*.

get_coupling_matrix()

Returns the currently stored mode-coupling matrix.

Returns mode-coupling matrix. The matrix will have shape *[nrows,nrows]*, with *nrows = n_cls * n_ells*, where *n_cls* is the number of power spectra (1, 2 or 4 for spin0-0, spin0-2 and spin2-2 correlations) and *n_ells = lmax + 1* (normally *lmax = 3 * nside - 1*). The assumed ordering of power spectra is such that the *l*-th element of the *i*-th power spectrum be stored with index *l * n_cls + i*.

read_from(*fname*)

Reads the contents of an NmtWorkspace object from a file (encoded using an internal binary format).

Parameters **fname** (*str*) – input file name

update_coupling_matrix(*new_matrix*)
Updates the stored mode-coupling matrix.

The new matrix (*new_matrix*) must have shape *[nrows,nrows]*, with *nrows = n_cls * n_ells*, where *n_cls* is the number of power spectra (1, 2 or 4 for spin0-0, spin0-2 and spin2-2 correlations) and *n_ells = lmax + 1* (normally *lmax = 3 * nside - 1*). The assumed ordering of power spectra is such that the *l*-th element of the *i*-th power spectrum be stored with index *l * n_cls + i*.

Parameters **new_matrix** – matrix that will replace the mode-coupling matrix.

write_to(*fname*)
Writes the contents of an NmtWorkspace object to a file (encoded using an internal binary format).

Parameters **fname** (*str*) – output file name

class pymaster.workspaces.**NmtWorkspaceFlat**
NmtWorkspaceFlat objects are used to compute and store the coupling matrix associated with an incomplete sky coverage, and used in the flat-sky version of the MASTER algorithm. When initialized, this object is practically empty. The information describing the coupling matrix must be computed or read from a file afterwards.

compute_coupling_matrix(*fl1, fl2, bins, ell_cut_x=[1.0, -1.0], ell_cut_y=[1.0, -1.0], is_teb=False*)
Computes coupling matrix associated with the cross-power spectrum of two NmtFieldFlats and an NmtBinFlat binning scheme.

Parameters

- **fl1,fl2** (NmtFieldFlat) – fields to correlate
- **bin** (NmtBinFlat) – binning scheme
- **ell_cut_x** (*float(2)*) – remove all modes with ell_x in the interval [ell_cut_x[0],ell_cut_x[1]] from the calculation.
- **ell_cut_y** (*float(2)*) – remove all modes with ell_y in the interval [ell_cut_y[0],ell_cut_y[1]] from the calculation.
- **is_teb** (*boolean*) – if true, all mode-coupling matrices (0-0,0-2,2-2) will be computed at the same time. In this case, fl1 must be a spin-0 field and fl1 must be spin-2.

couple_cell(*ells*, *cl_in*)
Convolves a set of input power spectra with a coupling matrix (see Eq. 6 of the C API documentation).

Parameters

- **ells** – list of multipoles on which the input power spectra are defined
- **cl_in** – set of input power spectra. The number of power spectra must correspond to the spins of the two fields that this NmtWorkspaceFlat object was initialized with (i.e. 1 for two spin-0 fields, 2 for one spin-0 and one spin-2 field and 4 for two spin-2 fields).

Returns coupled power spectrum. The coupled power spectra are returned at the multipoles returned by calling `get_ell_sampling()` for any of the fields that were used to generate the workspace.

decouple_cell(*cl_in*, *cl_bias=None*, *cl_noise=None*)
Decouples a set of pseudo-Cl power spectra into a set of bandpowers by inverting the binned coupling matrix (se Eq. 4 of the C API documentation).

Parameters

- **cl_in** – set of input power spectra. The number of power spectra must correspond to the spins of the two fields that this NmtWorkspaceFlat object was initialized with (i.e. 1 for two spin-0 fields, 2 for one spin-0 and one spin-2 field, 4 for two spin-2 fields and 7

if this NmtWorkspaceFlat was created using *is_teb=True*). These power spectra must be defined at the multipoles returned by `get_ell_sampling()` for any of the fields used to create the workspace.

- **cl_bias** – bias to the power spectrum associated to contaminant residuals (optional). This can be computed through `pymaster.deprojection_bias_flat()`.
- **cl_noise** – noise bias (i.e. angular power spectrum of masked noise realizations).

Returns set of decoupled bandpowers

read_from (*fname*)
Reads the contents of an NmtWorkspaceFlat object from a file (encoded using an internal binary format).

Parameters **fname** (*str*) – input file name

write_to (*fname*)
Writes the contents of an NmtWorkspaceFlat object to a file (encoded using an internal binary format).

Parameters **fname** (*str*) – output file name

pymaster.workspaces.**compute_coupled_cell** (*f1*, *f2*)
Computes the full-sky angular power spectra of two masked fields (f1 and f2) without aiming to deconvolve the mode-coupling matrix. Effectively, this is equivalent to calling the usual HEALPix anafast routine on the masked and contaminant-cleaned maps.

Parameters **f1,f2** (NmtField) – fields to correlate

Returns array of coupled power spectra

pymaster.workspaces.**compute_coupled_cell_flat** (*f1, f2, b, ell_cut_x=[1.0, -1.0], ell_cut_y=[1.0, -1.0]*)
Computes the angular power spectra of two masked flat-sky fields (f1 and f2) without aiming to deconvolve the mode-coupling matrix. Effectively, this is equivalent to computing the map FFTs and averaging over rings of wavenumber. The returned power spectrum is defined at the multipoles returned by the method `get_ell_sampling()` of either f1 or f2.

Parameters

- **f1,f2** (NmtFieldFlat) – fields to correlate
- **b** (NmtBinFlat) – binning scheme defining output bandpower
- **ell_cut_x** (*float(2)*) – remove all modes with ell_x in the interval [ell_cut_x[0],ell_cut_x[1]] from the calculation.
- **ell_cut_y** (*float(2)*) – remove all modes with ell_y in the interval [ell_cut_y[0],ell_cut_y[1]] from the calculation.

Returns array of coupled power spectra

pymaster.workspaces.**compute_full_master** (*f1, f2, b, cl_noise=None, cl_guess=None, workspace=None, n_iter=3, lmax_mask=-1*)
Computes the full MASTER estimate of the power spectrum of two fields (f1 and f2). This is equivalent to successively calling:

- `pymaster.NmtWorkspace.compute_coupling_matrix()`
- `pymaster.deprojection_bias()`
- `pymaster.compute_coupled_cell()`
- `pymaster.NmtWorkspace.decouple_cell()`

Parameters

- **f1,f2** (NmtField) – fields to correlate
- **b** (NmtBin) – binning scheme defining output bandpower
- **cl_noise** – noise bias (i.e. angular power spectrum of masked noise realizations) (optional).
- **cl_guess** – set of power spectra corresponding to a best-guess of the true power spectra of f1 and f2. Needed only to compute the contaminant cleaning bias (optional).
- **workspace** (NmtWorkspace) – object containing the mode-coupling matrix associated with an incomplete sky coverage. If provided, the function will skip the computation of the mode-coupling matrix and use the information encoded in this object.
- **n_iter** – number of iterations when computing a_lms.
- **lmax_mask** – maximum multipole for masks. If smaller than the maximum multipoles of the fields, it will be set to that.

Returns set of decoupled bandpowers

pymaster.workspaces.**compute_full_master_flat**(*f1, f2, b, cl_noise=None, cl_guess=None, ells_guess=None, workspace=None, ell_cut_x=[1.0, -1.0], ell_cut_y=[1.0, -1.0]*)

Computes the full MASTER estimate of the power spectrum of two flat-sky fields (f1 and f2). This is equivalent to successively calling:

- pymaster.NmtWorkspaceFlat.compute_coupling_matrix()
- pymaster.deprojection_bias_flat()
- pymaster.compute_coupled_cell_flat()
- pymaster.NmtWorkspaceFlat.decouple_cell()

Parameters

- **f1,f2** (NmtFieldFlat) – fields to correlate
- **b** (NmtBinFlat) – binning scheme defining output bandpower
- **cl_noise** – noise bias (i.e. angular power spectrum of masked noise realizations) (optional). This power spectrum should correspond to the bandpowers defined by b.
- **cl_guess** – set of power spectra corresponding to a best-guess of the true power spectra of f1 and f2. Needed only to compute the contaminant cleaning bias (optional).
- **ells_guess** – multipoles at which cl_guess is defined.
- **workspace** (NmtWorkspaceFlat) – object containing the mode-coupling matrix associated with an incomplete sky coverage. If provided, the function will skip the computation of the mode-coupling matrix and use the information encoded in this object.
- **nell_rebin** (*int*) – number of sub-intervals into which the base k-intervals will be subsampled to compute the coupling matrix
- **ell_cut_x** (*float(2)*) – remove all modes with ell_x in the interval [ell_cut_x[0],ell_cut_x[1]] from the calculation.
- **ell_cut_y** (*float(2)*) – remove all modes with ell_y in the interval [ell_cut_y[0],ell_cut_y[1]] from the calculation.

Returns set of decoupled bandpowers

pymaster.workspaces.**deprojection_bias**(*f1*, *f2*, *cls_guess*, *n_iter=3*)
Computes the bias associated to contaminant removal to the cross-pseudo-Cl of two fields.

Parameters

- **f1,f2** (NmtField) – fields to correlate
- **cls_guess** – set of power spectra corresponding to a best-guess of the true power spectra of f1 and f2.
- **n_iter** – number of iterations when computing a_lms.

Returns deprojection bias power spectra.

pymaster.workspaces.**deprojection_bias_flat**(*f1, f2, b, ells, cls_guess, ell_cut_x=[1.0, -1.0], ell_cut_y=[1.0, -1.0]*)
Computes the bias associated to contaminant removal to the cross-pseudo-Cl of two flat-sky fields. The returned power spectrum is defined at the multipoles returned by the method `get_ell_sampling()` of either f1 or f2.

Parameters

- **f1,f2** (NmtFieldFlat) – fields to correlate
- **b** (NmtBinFlat) – binning scheme defining output bandpower
- **ells** – list of multipoles on which the proposal power spectra are defined
- **cls_guess** – set of power spectra corresponding to a best-guess of the true power spectra of f1 and f2.
- **ell_cut_x** (*float(2)*) – remove all modes with ell_x in the interval [ell_cut_x[0],ell_cut_x[1]] from the calculation.
- **ell_cut_y** (*float(2)*) – remove all modes with ell_y in the interval [ell_cut_y[0],ell_cut_y[1]] from the calculation.

Returns deprojection bias power spectra.

pymaster.workspaces.**uncorr_noise_deprojection_bias**(*f1*, *map_var*, *n_iter=3*)
Computes the bias associated to contaminant removal in the presence of uncorrelated inhomogeneous noise to the auto-pseudo-Cl of a given field f1.

Parameters

- **f1** (NmtField) – fields to correlate
- **map_cls_guess** – array containing a HEALPix map corresponding to the local noise variance (in one sterad).
- **n_iter** – number of iterations when computing a_lms.

Returns deprojection bias power spectra.

class pymaster.covariance.**NmtCovarianceWorkspace**
NmtCovarianceWorkspace objects are used to compute and store the coupling coefficients needed to calculate the Gaussian covariance matrix under the approximations in astro-ph/0307515 and arXiv/1609.09730. When initialized, this object is practically empty. The information describing the coupling coefficients must be computed or read from a file afterwards.

compute_coupling_coefficients(*fla1*, *fla2*, *flb1=None*, *flb2=None*, *lmax=None*, *n_iter=3*)
Computes coupling coefficients of the Gaussian covariance between the power spectra of two pairs of NmtField objects (fla1, fla2, flb1 and flb2). Note that you can reuse this workspace for the covariance of power spectra between any pairs of fields as long as the fields have the same masks as those passed to this function, and as long as the binning scheme used are also the same.

Parameters

- **fla1,fla2** (NmtField) – fields contributing to the first power spectrum whose covariance you want to compute.
- **flb1,flb2** (NmtField) – fields contributing to the second power spectrum whose covariance you want to compute. If None, fla1,fla2 will be used.
- **n_iter** – number of iterations when computing a_lms.

read_from(*fname*)
Reads the contents of an NmtCovarianceWorkspace object from a file (encoded using an internal binary format).

Parameters **fname** (*str*) – input file name

write_to(*fname*)
Writes the contents of an NmtCovarianceWorkspace object to a file (encoded using an internal binary format).

Parameters **fname** (*str*) – output file name

class pymaster.covariance.**NmtCovarianceWorkspaceFlat**
NmtCovarianceWorkspaceFlat objects are used to compute and store the coupling coefficients needed to calculate the Gaussian covariance matrix under a flat-sky version the Efstathiou approximations in astro-ph/0307515 and arXiv/1609.09730. When initialized, this object is practically empty. The information describing the coupling coefficients must be computed or read from a file afterwards.

compute_coupling_coefficients(*fla1*, *fla2*, *bin_a*, *flb1=None*, *flb2=None*, *bin_b=None*)
Computes coupling coefficients of the Gaussian covariance between the power spectra of two pairs of NmtFieldFlat objects (fla1, fla2, flb1 and flb2). Note that you can reuse this workspace for the covariance of power spectra between any pairs of fields as long as the fields have the same masks as those passed to this function, and as long as the binning scheme used are also the same.

Parameters

- **fla1,fla2** (NmtFieldFlat) – fields contributing to the first power spectrum whose covariance you want to compute.
- **bin_a** (NmtBinFlat) – binning scheme for the first power spectrum.
- **flb1,flb2** (NmtFieldFlat) – fields contributing to the second power spectrum whose covariance you want to compute. If None, fla1,fla2 will be used.
- **bin_b** (NmtBinFlat) – binning scheme for the second power spectrum. If none, bin_a will be used.

read_from(*fname*)
Reads the contents of an NmtCovarianceWorkspaceFlat object from a file (encoded using an internal binary format).

Parameters **fname** (*str*) – input file name

write_to(*fname*)
Writes the contents of an NmtCovarianceWorkspaceFlat object to a file (encoded using an internal binary format).

Parameters **fname** (*str*) – output file name

pymaster.covariance.**gaussian_covariance**(*cw*, *spin_a1*, *spin_a2*, *spin_b1*, *spin_b2*, *cla1b1*, *cla1b2*, *cla2b1*, *cla2b2*, *wa*, *wb=None*)
Computes Gaussian covariance matrix for power spectra using the information precomputed in cw (a NmtCovarianceWorkspace object). cw should have been initialized using four NmtField objects (let's call them a1, a2, b1 and b2), corresponding to the two pairs of fields whose power spectra we want the covariance of. These

power spectra should have been computed using two NmtWorkspace objects, wa and wb, which must be passed as arguments of this function (the power spectrum for fields a1 and a2 was computed with wa, and that of b1 and b2 with wb). Using the same notation, clXnYm should be a prediction for the power spectrum between fields Xn and Ym. These predicted input power spectra should be defined for all ells <=3*nside (where nside is the HEALPix resolution parameter of the fields that were correlated).

Parameters

- **cw** (NmtCovarianceWorkspace) – workspaces containing the precomputed coupling coefficients.
- **spin_Xn** (*int*) – spin for the n-th field in pair X.
- **clXnYm** – prediction for the cross-power spectrum between fields Xn and Ym.
- **wX** (NmtWorkspace) – workspace containing the mode-coupling matrix pair X. If *wb* is *None*, the code will assume *wb=wa*.

pymaster.covariance.**gaussian_covariance_flat**(*cw*, *spin_a1*, *spin_a2*, *spin_b1*, *spin_b2*, *larr*, *cla1b1*, *cla1b2*, *cla2b1*, *cla2b2*, *wa*, *wb=None*)

Computes Gaussian covariance matrix for flat-sky power spectra using the information precomputed in cw (a NmtCovarianceWorkspaceFlat object). cw should have been initialized using four NmtFieldFlat objects (let's call them a1, a2, b1 and b2), corresponding to the two pairs of fields whose power spectra we want the covariance of. These power spectra should have been computed using two NmtWorkspaceFlat objects, wa and wb, which must be passed as arguments of this function (the power spectrum for fields a1 and a2 was computed with wa, and that of b1 and b2 with wb). Using the same notation, clXnYm should be a prediction for the power spectrum between fields Xn and Ym. These predicted input power spectra should be defined in a sufficiently well sampled range of ells given the map properties from which the power spectra were computed. The values of ell at which they are sampled are given by larr.

Please note that, while the method used to estimate these covariance is sufficiently accurate in a large number of scenarios, it is based on a numerical approximation, and its accuracy should be assessed if in doubt. In particular, we discourage users from using it to compute any covariance matrix involving B-mode components.

Parameters

- **cw** (NmtCovarianceWorkspaceFlat) – workspaces containing the precomputed coupling coefficients.
- **spin_Xn** (*int*) – spin for the n-th field in pair X.
- **larr** – values of ell at which the following power spectra are computed.
- **clXnYm** – prediction for the cross-power spectrum between fields Xn and Ym.
- **wX** (NmtWorkspaceFlat) – workspace containing the mode-coupling matrix pair X. If *wb* is *None*, the code will assume *wb=wa*.

class pymaster.utils.**NmtWCSTranslator**(*wcs*, *axes*)

This class takes care of interpreting a WCS object in terms of a Clenshaw-Curtis grid.

Parameters

- **wcs** – a WCS object (see http://docs.astropy.org/en/stable/wcs/index.html).
- **axes** – shape of the maps you want to analyze.

pymaster.utils.**mask_apodization**(*mask_in*, *aposize*, *apotype='C1'*)

Apodizes a mask with an given apodization scale using different methods.

Parameters

- **mask_in** – input mask, provided as an array of floats corresponding to a HEALPix map in RING order.
- **aposize** – apodization scale in degrees.
- **apotype** – apodization type. Three methods implemented: "C1", "C2" and "Smooth". See the description of the C-function nmt_apodize_mask in the C API documentation for a full description of these methods.

Returns apodized mask as a HEALPix map

pymaster.utils.**mask_apodization_flat**(*mask_in*, *lx*, *ly*, *aposize*, *apotype='C1'*)

Apodizes a flat-sky mask with an given apodization scale using different methods.

Parameters

- **mask_in** – input mask, provided as a 2D array (ny,nx) of floats.
- **lx** (*float*) – patch size in the x-axis (in radians)
- **ly** (*float*) – patch size in the y-axis (in radians)
- **aposize** – apodization scale in degrees.
- **apotype** – apodization type. Three methods implemented: "C1", "C2" and "Smooth". See the description of the C-function nmt_apodize_mask in the C API documentation for a full description of these methods.

Returns apodized mask as a 2D array (ny,nx)

pymaster.utils.**synfast_flat**(*nx*, *ny*, *lx*, *ly*, *cls*, *spin_arr*, *beam=None*, *seed=-1*)

Generates a flat-sky Gaussian random field according to a given power spectrum. This function is the flat-sky equivalent of healpy's synfast.

Parameters

- **nx** (*int*) – number of pixels in the x-axis
- **ny** (*int*) – number of pixels in the y-axis
- **lx** (*float*) – patch size in the x-axis (in radians)
- **ly** (*float*) – patch size in the y-axis (in radians)
- **cls** (*array-like*) – array containing power spectra. Shape should be [n_cls][n_ell], where n_cls is the number of power spectra needed to define all the fields. This should be n_cls = n_maps * (n_maps + 1) / 2, where n_maps is the total number of maps required (1 for each spin-0 field, 2 for each spin-2 field). Power spectra must be provided only for the upper-triangular part in row-major order (e.g. if n_maps is 3, there will be 6 power spectra ordered as [1-1,1-2,1-3,2-2,2-3,3-3].
- **spin_arr** (*array-like*) – array containing the spins of all the fields to generate.
- **array-like** (*beam*) – 2D array containing the instrumental beam of each field to simulate (the output map(s) will be convolved with it)
- **seed** (*int*) – RNG seed. If negative, will use a random seed.

Returns a number of arrays (1 for each spin-0 field, 2 for each spin-2 field) of size (ny,nx) containing the simulated maps.

pymaster.utils.**synfast_spherical**(*nside*, *cls*, *spin_arr*, *beam=None*, *seed=-1*, *wcs=None*)

Generates a full-sky Gaussian random field according to a given power spectrum. This function should produce outputs similar to healpy's synfast.

Parameters

- **nside** (*int*) – HEALpix resolution parameter. If you want rectangular pixel maps, ignore this parameter and pass a WCS object as *wcs* (see below).
- **cls** (*array-like*) – array containing power spectra. Shape should be [n_cls][n_ell], where n_cls is the number of power spectra needed to define all the fields. This should be n_cls = n_maps * (n_maps + 1) / 2, where n_maps is the total number of maps required (1 for each spin-0 field, 2 for each spin-2 field). Power spectra must be provided only for the upper-triangular part in row-major order (e.g. if n_maps is 3, there will be 6 power spectra ordered as [1-1,1-2,1-3,2-2,2-3,3-3].
- **spin_arr** (*array-like*) – array containing the spins of all the fields to generate.
- **array-like** (*beam*) – 2D array containing the instrumental beam of each field to simulate (the output map(s) will be convolved with it)
- **seed** (*int*) – RNG seed. If negative, will use a random seed.
- **wcs** – a WCS object (see http://docs.astropy.org/en/stable/wcs/index.html).

Returns a number of full-sky maps (1 for each spin-0 field, 2 for each spin-2 field).

CHAPTER 2

Example 1: simple pseudo-Cl computation

This sample script shows the simplest way to compute the cross-power spectrum between two fields

```
import numpy as np
import healpy as hp
import matplotlib.pyplot as plt

# Import the NaMaster python wrapper
import pymaster as nmt

#  Simple example showcasing the use of NaMaster to compute the pseudo-Cl
#  estimator of the angular cross-power spectrum of a spin-0 field and a
#  spin-2 field

# HEALPix resolution parameter used here
nside = 256

# Read mask and apodize it on a scale of ~1deg
mask = nmt.mask_apodization(hp.read_map("mask.fits", verbose=False),
                            1., apotype="Smooth")
hp.mollview(mask, coord=['G', 'C'], title='Apodized mask')
plt.show()

# Read healpix maps and initialize a spin-0 and spin-2 field
f_0 = nmt.NmtField(mask, [hp.read_map("maps.fits", field=0, verbose=False)])
f_2 = nmt.NmtField(mask, hp.read_map("maps.fits", field=[1, 2], verbose=False))

# Initialize binning scheme with 4 ells per bandpower
b = nmt.NmtBin.from_nside_linear(nside, 4)

# Compute MASTER estimator
# spin-0 x spin-0
cl_00 = nmt.compute_full_master(f_0, f_0, b)
# spin-0 x spin-2
cl_02 = nmt.compute_full_master(f_0, f_2, b)
```

(continues on next page)

(continued from previous page)

```
# spin-2 x spin-2
cl_22 = nmt.compute_full_master(f_2, f_2, b)

# Plot results
ell_arr = b.get_effective_ells()
plt.plot(ell_arr, cl_00[0], 'r-', label='TT')
plt.plot(ell_arr, np.fabs(cl_02[0]), 'g-', label='TE')
plt.plot(ell_arr, cl_22[0], 'b-', label='EE')
plt.plot(ell_arr, cl_22[3], 'y-', label='BB')
plt.loglog()
plt.xlabel('$\\ell$', fontsize=16)
plt.ylabel('$C_\\ell$', fontsize=16)
plt.legend(loc='upper right', ncol=2, labelspacing=0.1)
plt.show()
```

CHAPTER 3

Example 2: Bandpowers

This sample script showcases the use of the NmtBin class to define bandpowers.

```
import numpy as np
import matplotlib.pyplot as plt
import pymaster as nmt

# This script showcases the use of the NmtBin structure to define bandpowers.

# HEALPix map resolution
nside = 256

# Initialize binning scheme with bandpowers of constant width
# (4 multipoles per bin)
bin1 = nmt.NmtBin.from_nside_linear(nside, 4)

# Initialize binning scheme with custom-made bandpowers.
# In this case we simply manually choose these bandpowers to also have
# 4 multipoles per bin.
ells = np.arange(3 * nside, dtype='int32')  # Array of multipoles
weights = 0.25 * np.ones_like(ells)  # Array of weights
bpws = -1 + np.zeros_like(ells)  # Array of bandpower indices
i = 0
while 4 * (i + 1) + 2 < 3 * nside:
    bpws[4 * i + 2:4 * (i + 1) + 2] = i
    i += 1
bin2 = nmt.NmtBin(nside=nside, bpws=bpws, ells=ells, weights=weights)

# You can also control ell-weighting through NmtBins.
# E.g. to compute the usual D_ell = ell * (ell + 1) * C_ell/2/pi,
# you can use is_Dell=True
bin3 = nmt.NmtBin.from_nside_linear(nside, 4, is_Dell=True)

# At this stage bin1 and bin2 should be identical
print(np.sum(bin1.get_effective_ells()-bin2.get_effective_ells()))
```

(continues on next page)

(continued from previous page)

```
# Array with effective multipole per bandpower
ell_eff = bin1.get_effective_ells()

# Bandpower info:
print("Bandpower info:")
print(" %d bandpowers" % (bin1.get_n_bands()))
print("The columns in the following table are:")
print(" [1]-band index, [2]-list of multipoles, "
      "[3]-list of weights, [4]=effective multipole")
for i in range(bin1.get_n_bands()):
    print(i, bin1.get_ell_list(i), bin1.get_weight_list(i), ell_eff[i])
print("")

# Binning a power spectrum
# Read the TT power spectrum
data = np.loadtxt("cls.txt", unpack=True)
ell_arr = data[0]
cl_tt = data[1]
# Bin the power spectrum into bandpowers
cl_tt_binned = bin1.bin_cell(np.array([cl_tt]))
# For bin3 we need to correct for the ell prefactor
ellfac = ell_eff * (ell_eff + 1.)/2/np.pi
dl_tt_binned = bin3.bin_cell(np.array([cl_tt]))/ellfac
# Unbin bandpowers
cl_tt_binned_unbinned = bin1.unbin_cell(cl_tt_binned)
# Plot all to see differences
plt.plot(ell_arr, cl_tt, 'r-',
         label='Original $C_\\ell$')
plt.plot(ell_eff, cl_tt_binned[0], 'g-',
         label='Binned $C_\\ell$')
plt.plot(ell_eff, dl_tt_binned[0], 'y-',
         label='Binned $D_\\ell 2\\pi/(\\ell(\\ell+1))$')
plt.plot(ell_arr, cl_tt_binned_unbinned[0], 'b-',
         label='Binned-unbinned $C_\\ell$')
plt.loglog()
plt.legend(loc='upper right', frameon=False)
plt.show()
```

CHAPTER 4

Example 3: Fields

This sample script showcases the use of the NmtField class to define and use observed fields.

```
import healpy as hp
import matplotlib.pyplot as plt
import pymaster as nmt

# This script showcases the use of the NmtField structure to store information
# about the fields to be correlated.

# HEALPix map resolution
nside = 256

# # # #  Read input maps
# a) Read and apodize mask
mask = nmt.mask_apodization(hp.read_map("mask.fits", verbose=False),
                            1., apotype="C1")
# b) Read maps
mp_t, mp_q, mp_u = hp.read_map("maps.fits", field=[0, 1, 2], verbose=False)
# c) Read contaminants maps
tm_t, tm_q, tm_u = hp.read_map("temp.fits", field=[0, 1, 2], verbose=False)

# Create fields
# Create spin-0 field with no contaminants
f0_clean = nmt.NmtField(mask, [mp_t])
# Create spin-2 field with no contaminants
f2_clean = nmt.NmtField(mask, [mp_q, mp_u])
# Create contaminated spin-0 field
f0_cont = nmt.NmtField(mask, [mp_t+tm_t], templates=[[tm_t]])
# Create contaminated spin-2 field
f2_cont = nmt.NmtField(mask, [mp_q+tm_q, mp_u+tm_u], templates=[[tm_q, tm_u]])

# Note: when passing "templates", the constructor cleans the maps by finding
# the best-fit linear coefficient that fits the contaminant templates.
# I.e. the maps returned by, e.g. f0_cont.get_maps(), are already cleaned.
```

(continues on next page)

(continued from previous page)

```
#  - f0_clean and f2_clean now hold masked copies of the original maps.
#  - f0_cont and f2_cont now hold masked and ***template-cleaned*** copies of
#    the original maps

# We can verify this by plotting them:
# Spin-0
hp.mollview(mp_t, title='Original map T', coord=['G', 'C'])
hp.mollview(mp_t+tm_t, title='Contaminated map T', coord=['G', 'C'])
hp.mollview(f0_clean.get_maps()[0],
            title='Masked original map T', coord=['G', 'C'])
hp.mollview(f0_cont.get_maps()[0],
            title='Masked & cleaned map T', coord=['G', 'C'])
plt.show()
# Spin-2, Q
hp.mollview(mp_q, title='Original map Q', coord=['G', 'C'])
hp.mollview(mp_q+tm_q, title='Contaminated map Q', coord=['G', 'C'])
hp.mollview(f2_clean.get_maps()[0],
            title='Masked original map Q', coord=['G', 'C'])
hp.mollview(f2_cont.get_maps()[0],
            title='Masked & cleaned map Q', coord=['G', 'C'])
plt.show()
# Spin-2, U
hp.mollview(mp_u, title='Original map U', coord=['G', 'C'])
hp.mollview(mp_u+tm_u, title='Contaminated map U', coord=['G', 'C'])
hp.mollview(f2_clean.get_maps()[1],
            title='Masked original map U', coord=['G', 'C'])
hp.mollview(f2_cont.get_maps()[1],
            title='Masked & cleaned map U', coord=['G', 'C'])
plt.show()
```

CHAPTER 5

Example 4: Masks

This sample script showcases the apodization routine implemented in NaMaster

```
import healpy as hp
import matplotlib.pyplot as plt
import pymaster as nmt

# This script showcases the apodization routine included with pymaster
# and the three apodization modes supported.


# Read input binary mask
mask_raw = hp.read_map("mask.fits", verbose=False)

# The following function calls create apodized versions of the raw mask
# with an apodization scale of 2.5 degrees using three different methods

# Apodization scale in degrees
aposcale = 2.5

# C1 and C2: in these cases, pixels are multiplied by a factor f
#            (with 0<=f<=1) based on their distance to the nearest fully
#            masked pixel. The choices of f in each case are documented in
#            Section 3.4 of the C API documentation. All pixels separated
#            from any masked pixel by more than the apodization scale are
#            left untouched.
mask_C1 = nmt.mask_apodization(mask_raw, aposcale, apotype="C1")
mask_C2 = nmt.mask_apodization(mask_raw, aposcale, apotype="C2")

# Smooth: in this case, all pixels closer to a masked pixel than 2.5 times
#         the apodization scale are initially set to zero. The resulting
#         map is then smoothed with a Gaussian kernel with standard
#         deviation given by the apodization scale. Finally, all pixels
#         originally masked are forced back to zero.
mask_Sm = nmt.mask_apodization(mask_raw, aposcale, apotype="Smooth")
```

(continues on next page)

(continued from previous page)

```
# Let's plot the results
hp.mollview(mask_raw, title='Binary mask', coord=['G', 'C'])
hp.mollview(mask_C1, title='C1 apodization', coord=['G', 'C'])
hp.mollview(mask_C2, title='C2 apodization', coord=['G', 'C'])
hp.mollview(mask_Sm, title='Smooth apodization', coord=['G', 'C'])
plt.show()
```

CHAPTER 6

Example 5: Using workspaces

This sample script showcases the use of the NmtWorkspace class to speed up the computation of multiple power spectra with the same mask. This is the most general example in this suite, showing also the correct way to compare the results of the MASTER estimator with the theory power spectrum.

```
import numpy as np
import healpy as hp
import matplotlib.pyplot as plt
import pymaster as nmt

# This script showcases the use of NmtWorkspace objects to speed up the
# computation of power spectra for many pairs of fields with the same masks.

# HEALPix map resolution
nside = 256

# We start by creating some synthetic masks and maps with contaminants.
# Here we will focus on the cross-correlation of a spin-2 and a spin-1 field.
# a) Read and apodize mask
mask = nmt.mask_apodization(hp.read_map("mask.fits", verbose=False),
                            1., apotype="Smooth")
# b) Read maps
mp_t, mp_q, mp_u = hp.read_map("maps.fits", field=[0, 1, 2], verbose=False)
# c) Read contaminants maps
tm_t, tm_q, tm_u = hp.read_map("temp.fits", field=[0, 1, 2], verbose=False)
# d) Create contaminated fields
#    Spin-0
f0 = nmt.NmtField(mask, [mp_t+tm_t], templates=[[tm_t]])
#    Spin-2
f2 = nmt.NmtField(mask, [mp_q+tm_q, mp_u+tm_u], templates=[[tm_q, tm_u]])
# e) Create binning scheme. We will use 20 multipoles per bandpower.
b = nmt.NmtBin.from_nside_linear(nside, 20)
# Note that, if you want to compute bandpowers assuming constant underlying
# D_ell = ell*(ell+1)*C_ell/(2*pi) (instead of constant C_ell) within each
# bandpower, you can add `is_Dell=True` when defining an NmtBin. The
```

(continues on next page)

(continued from previous page)

```
# prefactor is actually fully tuneable - check out the documentation for the
# NmtBin constructor.
#
# f) Finally, we read our best guess for the true power spectrum. We will
#    use this to:
#    i)  Compute the bias to the power spectrum from contaminant cleaning
#    ii) Generate random realizations of our fields to compute the errors
l, cltt, clee, clbb, clte = np.loadtxt("cls.txt", unpack=True)
cl_02_th = np.array([clte, np.zeros_like(clte)])

# We then generate an NmtWorkspace object that we use to compute and store
# the mode coupling matrix. Note that this matrix depends only on the masks
# of the two fields to correlate, but not on the maps themselves (in this
# case both maps are the same.
w = nmt.NmtWorkspace()
w.compute_coupling_matrix(f0, f2, b)

# Since we suspect that our maps are contaminated (that's why we passed the
# contaminant templates as arguments to the NmtField constructor), we also
# need to compute the bias to the power spectrum caused by contaminant
# cleaning (deprojection bias).
cl_bias = nmt.deprojection_bias(f0, f2, cl_02_th)


# The function defined below will compute the power spectrum between two
# NmtFields f_a and f_b, using the coupling matrix stored in the
# NmtWorkspace wsp and subtracting the deprojection bias clb.
# Note that the most expensive operations in the MASTER algorithm are
# the computation of the coupling matrix and the deprojection bias. Since
# these two objects are precomputed, this function should be pretty fast!
def compute_master(f_a, f_b, wsp, clb):
    # Compute the power spectrum (a la anafast) of the masked fields
    # Note that we only use n_iter=0 here to speed up the computation,
    # but the default value of 3 is recommended in general.
    cl_coupled = nmt.compute_coupled_cell(f_a, f_b)
    # Decouple power spectrum into bandpowers inverting the coupling matrix
    cl_decoupled = wsp.decouple_cell(cl_coupled, cl_bias=clb)

    return cl_decoupled


# OK, we can now compute the power spectrum of our two input fields
cl_master = compute_master(f0, f2, w, cl_bias)

# Let's now compute the errors on this estimator using 100 Gaussian random
# simulations. In a realistic scenario you'd want to compute the full
# covariance matrix, but let's keep things simple.
nsim = 100
cl_mean = np.zeros_like(cl_master)
cl_std = np.zeros_like(cl_master)
for i in np.arange(nsim):
    print("%d-th simulation" % i)
    t, q, u = hp.synfast([cltt, clee, clbb, clte], nside, verbose=False)
    f0_sim = nmt.NmtField(mask, [t], templates=[[tm_t]])
    f2_sim = nmt.NmtField(mask, [q, u], templates=[[tm_q, tm_u]])
    cl = compute_master(f0_sim, f2_sim, w, cl_bias)
    cl_mean += cl
```

(continues on next page)

(continued from previous page)

```
    cl_std += cl*cl
cl_mean /= nsim
cl_std = np.sqrt(cl_std / nsim - cl_mean*cl_mean)

# One final thing needs to be done before we can compare the result with
# the theory. The theory power spectrum must be binned into bandpowers in
# the same manner the data has. This is straightforward to do using just
# two nested function calls.
cl_02_th_binned = w.decouple_cell(w.couple_cell(cl_02_th))

# Now let's plot the result!
plt.plot(b.get_effective_ells(), cl_02_th_binned[0], 'r-',
         label='True power spectrum')
plt.plot(b.get_effective_ells(), cl_02_th_binned[1], 'g-')
plt.errorbar(b.get_effective_ells(), cl_master[0], yerr=cl_std[0],
             fmt='ro', label='MASTER estimate (TE)')
plt.errorbar(b.get_effective_ells(), cl_master[1], yerr=cl_std[1],
             fmt='bo', label='MASTER estimate (TB)')
plt.ylim([-0.03, 0.03])
plt.legend(loc='upper right')
plt.xlabel('$\\ell$', fontsize=16)
plt.ylabel('$C_\\ell$', fontsize=16)
plt.show()
```

CHAPTER 7

Example 6: Pure E and B

This sample script showcases the computation of power spectra using the pure-E and B approach.

Warning: If you have HEALPix UNSEEN values in your map, set those pixels to zero in both the mask and map in order to use B-mode purification.

```
import numpy as np
import healpy as hp
import matplotlib.pyplot as plt
import pymaster as nmt

# This script describes the computation of polarized power spectra using the
# pure-E and B approach

# We'll run this many simulations
nsim = 10
# HEALPix map resolution
nside = 256

# Let us first create a square mask:
msk = np.zeros(hp.nside2npix(nside))
th, ph = hp.pix2ang(nside, np.arange(hp.nside2npix(nside)))
ph[np.where(ph > np.pi)[0]] -= 2 * np.pi
msk[np.where((th < 2.63) & (th > 1.86) &
             (ph > -np.pi / 4) & (ph < np.pi / 4))[0]] = 1.

# Now we apodize the mask. The pure-B formalism requires the mask to be
# differentiable along the edges. The 'C1' and 'C2' apodization types
# supported by mask_apodization achieve this.
msk_apo = nmt.mask_apodization(msk, 10.0, apotype='C1')

# Select a binning scheme
b = nmt.NmtBin.from_nside_linear(nside, 16)
```

(continues on next page)

(continued from previous page)

```
leff = b.get_effective_ells()

# Read power spectrum and provide function to generate simulated skies
l, cltt, clee, clbb, clte = np.loadtxt('cls.txt', unpack=True)


def get_fields():
    mp_t, mp_q, mp_u = hp.synfast([cltt, clee, clbb, clte],
                                  nside=nside, new=True, verbose=False)
    # This creates a spin-2 field without purifying either E or B
    f2_np = nmt.NmtField(msk_apo, [mp_q, mp_u])
    # This creates a spin-2 field with both pure E and B.
    f2_yp = nmt.NmtField(msk_apo, [mp_q, mp_u], purify_e=True, purify_b=True)
    # Note that generally it's not a good idea to purify both,
    # since you'll lose sensitivity on E
    return f2_np, f2_yp


# We initialize two workspaces for the non-pure and pure fields:
f2np0, f2yp0 = get_fields()
w_np = nmt.NmtWorkspace()
w_np.compute_coupling_matrix(f2np0, f2np0, b)
w_yp = nmt.NmtWorkspace()
w_yp.compute_coupling_matrix(f2yp0, f2yp0, b)


# This wraps up the two steps needed to compute the power spectrum
# once the workspace has been initialized
def compute_master(f_a, f_b, wsp):
    cl_coupled = nmt.compute_coupled_cell(f_a, f_b)
    cl_decoupled = wsp.decouple_cell(cl_coupled)
    return cl_decoupled


# We now iterate over several simulations, computing the
# power spectrum for each of them
data_np = []
data_yp = []
for i in np.arange(nsim):
    print(i, nsim)
    fnp, fyp = get_fields()
    data_np.append(compute_master(fnp, fnp, w_np))
    data_yp.append(compute_master(fyp, fyp, w_yp))
data_np = np.array(data_np)
data_yp = np.array(data_yp)
clnp_mean = np.mean(data_np, axis=0)
clnp_std = np.std(data_np, axis=0)
clyp_mean = np.mean(data_yp, axis=0)
clyp_std = np.std(data_yp, axis=0)

# Now we plot the results
plt.figure()
plt.title('$BB$ error', fontsize=18)
plt.plot(leff, clnp_std[3], 'r-', lw=2, label='Standard pseudo-$C_\\ell$')
plt.plot(leff, clyp_std[3], 'b-', lw=2, label='Pure-$B$ estimator')
plt.xlim([2, 512])
plt.xlabel('$\\ell$', fontsize=18)
```

(continues on next page)

(continued from previous page)

```
plt.ylabel('$\\sigma(C_\\ell)$', fontsize=18)
plt.legend(loc='upper right', frameon=False)
plt.loglog()
plt.show()
```

CHAPTER 8

Example 7: Flat-sky fields

This sample script showcases the flat-sky version of pymaster.

```
import numpy as np
import pymaster as nmt
import matplotlib.pyplot as plt


# This script describes the functionality of the flat-sky version of pymaster

# Dimensions:
# First, a flat-sky field is defined by four quantities:
#  - Lx and Ly: the size of the patch in the x and y dimensions (in radians)
Lx = 72. * np.pi/180
Ly = 48. * np.pi/180
#  - Nx and Ny: the number of pixels in the x and y dimensions
Nx = 602
Ny = 410

# Gaussian simulations:
# pymaster allows you to generate random realizations of both spherical and
# flat fields given a power spectrum. These are returned as 2D arrays with
# shape (Ny,Nx)
l, cl_tt, cl_ee, cl_bb, cl_te = np.loadtxt('cls.txt', unpack=True)
beam = np.exp(-(0.25 * np.pi/180 * l)**2)
cl_tt *= beam
cl_ee *= beam
cl_bb *= beam
cl_te *= beam
mpt, mpq, mpu = nmt.synfast_flat(Nx, Ny, Lx, Ly,
                                 np.array([cl_tt, cl_te, 0 * cl_tt,
                                           cl_ee, 0 * cl_ee, cl_bb]),
                                 [0, 2])

# You can have a look at the maps using matplotlib's imshow:
```

(continues on next page)

(continued from previous page)

```
plt.figure()
plt.imshow(mpt, interpolation='nearest', origin='lower')
plt.colorbar()
plt.figure()
plt.imshow(mpq, interpolation='nearest', origin='lower')
plt.colorbar()
plt.figure()
plt.imshow(mpu, interpolation='nearest', origin='lower')
plt.colorbar()
plt.show()

# Masks:
# Let's now create a mask:
mask = np.ones_like(mpt).flatten()
xarr = np.ones(Ny)[:, None] * np.arange(Nx)[None, :] * Lx/Nx
yarr = np.ones(Nx)[None, :] * np.arange(Ny)[:, None] * Ly/Ny


# First we dig a couple of holes
def dig_hole(x, y, r):
    rad = (np.sqrt((xarr - x)**2 + (yarr - y)**2)).flatten()
    return np.where(rad < r)[0]


mask[dig_hole(0.3 * Lx, 0.6 * Ly, 0.05 * np.sqrt(Lx * Ly))] = 0.
mask[dig_hole(0.7 * Lx, 0.12 * Ly, 0.07 * np.sqrt(Lx * Ly))] = 0.
mask[dig_hole(0.7 * Lx, 0.8 * Ly, 0.03 * np.sqrt(Lx * Ly))] = 0.
# Let's also trim the edges
mask[np.where(xarr.flatten() < Lx / 16.)] = 0
mask[np.where(xarr.flatten() > 15 * Lx / 16.)] = 0
mask[np.where(yarr.flatten() < Ly / 16.)] = 0
mask[np.where(yarr.flatten() > 15 * Ly / 16.)] = 0
mask = mask.reshape([Ny, Nx])
# You can also apodize it in the same way you do for full-sky masks:
mask = nmt.mask_apodization_flat(mask, Lx, Ly, aposize=2., apotype="C1")
plt.figure()
plt.imshow(mask, interpolation='nearest', origin='lower')
plt.colorbar()
plt.show()

# Fields:
# Once you have maps it's time to create pymaster fields.
# Note that, as in the full-sky case, you can also pass
# contaminant templates and flags for E and B purification
# (see the documentation for more details)
f0 = nmt.NmtFieldFlat(Lx, Ly, mask, [mpt])
f2 = nmt.NmtFieldFlat(Lx, Ly, mask, [mpq, mpu], purify_b=True)

# Bins:
# For flat-sky fields, bandpowers are simply defined as intervals in ell, and
# pymaster doesn't currently support any weighting scheme within each interval.
l0_bins = np.arange(Nx/8) * 8 * np.pi/Lx
lf_bins = (np.arange(Nx/8)+1) * 8 * np.pi/Lx
b = nmt.NmtBinFlat(l0_bins, lf_bins)
# The effective sampling rate for these bandpowers can be obtained calling:
ells_uncoupled = b.get_effective_ells()
```

(continues on next page)

(continued from previous page)

```
# Workspaces:
# As in the full-sky case, the computation of the coupling matrix and of
# the pseudo-CL estimator is mediated by a WorkspaceFlat case, initialized
# by calling its compute_coupling_matrix method:
w00 = nmt.NmtWorkspaceFlat()
w00.compute_coupling_matrix(f0, f0, b)
w02 = nmt.NmtWorkspaceFlat()
w02.compute_coupling_matrix(f0, f2, b)
w22 = nmt.NmtWorkspaceFlat()
w22.compute_coupling_matrix(f2, f2, b)
# Workspaces can be saved to and read from disk to avoid recomputing them:
w00.write_to("w00_flat.dat")
w00.read_from("w00_flat.dat")
w02.write_to("w02_flat.dat")
w02.read_from("w02_flat.dat")
w22.write_to("w22_flat.dat")
w22.read_from("w22_flat.dat")

# Computing power spectra:
# As in the full-sky case, you compute the pseudo-CL estimator by
# computing the coupled power spectra and then decoupling them by
# inverting the mode-coupling matrix. This is done in two steps below,
# but pymaster provides convenience routines to do this
# through a single function call
cl00_coupled = nmt.compute_coupled_cell_flat(f0, f0, b)
cl00_uncoupled = w00.decouple_cell(cl00_coupled)
cl02_coupled = nmt.compute_coupled_cell_flat(f0, f2, b)
cl02_uncoupled = w02.decouple_cell(cl02_coupled)
cl22_coupled = nmt.compute_coupled_cell_flat(f2, f2, b)
cl22_uncoupled = w22.decouple_cell(cl22_coupled)

# Let's look at the results!
plt.figure()
plt.plot(l, cl_tt, 'r-', label='Input TT')
plt.plot(l, cl_ee, 'g-', label='Input EE')
plt.plot(l, cl_bb, 'b-', label='Input BB')
plt.plot(ells_uncoupled, cl00_uncoupled[0], 'r--', label='Uncoupled')
plt.plot(ells_uncoupled, cl22_uncoupled[0], 'g--')
plt.plot(ells_uncoupled, cl22_uncoupled[3], 'b--')
plt.loglog()
plt.show()
```

CHAPTER 9

Example 8: Computing covariance matrices

This sample script showcases the ability of NaMaster to estimate the Gaussian covariance matrix for the pseudo-Cl estimator.

```
import numpy as np
import healpy as hp
import matplotlib.pyplot as plt
import pymaster as nmt

# This script showcases the ability of namaster to compute Gaussian
# estimates of the covariance matrix.
# A similar example for flat-sky fields can be found in
# test/sample_covariance_flat.py

# HEALPix map resolution
nside = 256

# We start by creating some synthetic masks and maps with contaminants.
# Here we will focus on the auto-correlation of a spin-1 field.
# a) Read and apodize mask
mask = nmt.mask_apodization(hp.read_map("mask.fits", verbose=False),
                            1., apotype="Smooth")

# Let's now create a fictitious theoretical power spectrum to generate
# Gaussian realizations:
larr = np.arange(3*nside)
clarr = ((larr+1.)/80.)**(-1.1)+1.
cl_tt = clarr
cl_ee = clarr
cl_bb = 0*clarr
cl_te = 0*clarr
cl_tb = 0*clarr
cl_eb = 0*clarr
```

(continues on next page)

(continued from previous page)

```
# This routine generates a spin-0 and a spin-2 Gaussian random field based
# on these power spectra
def get_sample_field():
    mp_t, mp_q, mp_u = hp.synfast([cl_tt, cl_ee, cl_bb, cl_te],
                                  nside, verbose=False)
    return nmt.NmtField(mask, [mp_t]), nmt.NmtField(mask, [mp_q, mp_u])


# We also copy this function from sample_workspaces.py. It computes
# power spectra given a pair of fields and a workspace.
def compute_master(f_a, f_b, wsp):
    cl_coupled = nmt.compute_coupled_cell(f_a, f_b)
    cl_decoupled = wsp.decouple_cell(cl_coupled)

    return cl_decoupled


# Let's generate one particular sample and its power spectrum.
print("Field")
f0, f2 = get_sample_field()
# We will use 20 multipoles per bandpower.
b = nmt.NmtBin.from_nside_linear(nside, 20)
print("Workspace")
w00 = nmt.NmtWorkspace()
w00.compute_coupling_matrix(f0, f0, b)
w02 = nmt.NmtWorkspace()
w02.compute_coupling_matrix(f0, f2, b)
w22 = nmt.NmtWorkspace()
w22.compute_coupling_matrix(f2, f2, b)
cl_00 = compute_master(f0, f0, w00)
cl_02 = compute_master(f0, f2, w02)
cl_22 = compute_master(f2, f2, w22)
n_ell = len(cl_00[0])

# Let's now compute the Gaussian estimate of the covariance!
print("Covariance")
# First we generate a NmtCovarianceWorkspace object to precompute
# and store the necessary coupling coefficients
cw = nmt.NmtCovarianceWorkspace()
# This is the time-consuming operation
# Note that you only need to do this once,
# regardless of spin
cw.compute_coupling_coefficients(f0, f0, f0, f0)

# The next few lines show how to extract the covariance matrices
# for different spin combinations.
covar_00_00 = nmt.gaussian_covariance(cw,
                                      0, 0, 0, 0,  # Spins of the 4 fields
                                      [cl_tt],  # TT
                                      [cl_tt],  # TT
                                      [cl_tt],  # TT
                                      [cl_tt],  # TT
                                      w00, wb=w00).reshape([n_ell, 1,
                                                            n_ell, 1])
covar_TT_TT = covar_00_00[:, 0, :, 0]
covar_02_02 = nmt.gaussian_covariance(cw, 0, 2, 0, 2,  # Spins of the 4 fields
                                      [cl_tt],  # TT
```

(continues on next page)

(continued from previous page)

```
                                      [cl_te, cl_tb],  # TE, TB
                                      [cl_te, cl_tb],  # ET, BT
                                      [cl_ee, cl_eb,
                                       cl_eb, cl_bb],  # EE, EB, BE, BB
                                      w02, wb=w02).reshape([n_ell, 2,
                                                            n_ell, 2])
covar_TE_TE = covar_02_02[:, 0, :, 0]
covar_TE_TB = covar_02_02[:, 0, :, 1]
covar_TB_TE = covar_02_02[:, 1, :, 0]
covar_TB_TB = covar_02_02[:, 1, :, 1]


covar_00_22 = nmt.gaussian_covariance(cw, 0, 0, 2, 2,  # Spins of the 4 fields
                                      [cl_te, cl_tb],  # TE, TB
                                      [cl_te, cl_tb],  # TE, TB
                                      [cl_te, cl_tb],  # TE, TB
                                      [cl_te, cl_tb],  # TE, TB
                                      w00, wb=w22).reshape([n_ell, 1,
                                                            n_ell, 4])
covar_TT_EE = covar_00_22[:, 0, :, 0]
covar_TT_EB = covar_00_22[:, 0, :, 1]
covar_TT_BE = covar_00_22[:, 0, :, 2]
covar_TT_BB = covar_00_22[:, 0, :, 3]

covar_02_22 = nmt.gaussian_covariance(cw, 0, 2, 2, 2,  # Spins of the 4 fields
                                      [cl_te, cl_tb],  # TE, TB
                                      [cl_te, cl_tb],  # TE, TB
                                      [cl_ee, cl_eb,
                                       cl_eb, cl_bb],  # EE, EB, BE, BB
                                      [cl_ee, cl_eb,
                                       cl_eb, cl_bb],  # EE, EB, BE, BB
                                      w02, wb=w22).reshape([n_ell, 2,
                                                            n_ell, 4])
covar_TE_EE = covar_02_22[:, 0, :, 0]
covar_TE_EB = covar_02_22[:, 0, :, 1]
covar_TE_BE = covar_02_22[:, 0, :, 2]
covar_TE_BB = covar_02_22[:, 0, :, 3]
covar_TB_EE = covar_02_22[:, 1, :, 0]
covar_TB_EB = covar_02_22[:, 1, :, 1]
covar_TB_BE = covar_02_22[:, 1, :, 2]
covar_TB_BB = covar_02_22[:, 1, :, 3]


covar_22_22 = nmt.gaussian_covariance(cw, 2, 2, 2, 2,  # Spins of the 4 fields
                                      [cl_ee, cl_eb,
                                       cl_eb, cl_bb],  # EE, EB, BE, BB
                                      [cl_ee, cl_eb,
                                       cl_eb, cl_bb],  # EE, EB, BE, BB
                                      [cl_ee, cl_eb,
                                       cl_eb, cl_bb],  # EE, EB, BE, BB
                                      [cl_ee, cl_eb,
                                       cl_eb, cl_bb],  # EE, EB, BE, BB
                                      w22, wb=w22).reshape([n_ell, 4,
                                                            n_ell, 4])
covar_EE_EE = covar_22_22[:, 0, :, 0]
covar_EE_EB = covar_22_22[:, 0, :, 1]
covar_EE_BE = covar_22_22[:, 0, :, 2]
```

(continues on next page)

(continued from previous page)

```
covar_EE_BB = covar_22_22[:, 0, :, 3]
covar_EB_EE = covar_22_22[:, 1, :, 0]
covar_EB_EB = covar_22_22[:, 1, :, 1]
covar_EB_BE = covar_22_22[:, 1, :, 2]
covar_EB_BB = covar_22_22[:, 1, :, 3]
covar_BE_EE = covar_22_22[:, 2, :, 0]
covar_BE_EB = covar_22_22[:, 2, :, 1]
covar_BE_BE = covar_22_22[:, 2, :, 2]
covar_BE_BB = covar_22_22[:, 2, :, 3]
covar_BB_EE = covar_22_22[:, 3, :, 0]
covar_BB_EB = covar_22_22[:, 3, :, 1]
covar_BB_BE = covar_22_22[:, 3, :, 2]
covar_BB_BB = covar_22_22[:, 3, :, 3]

# Let's now compute the sample covariance
# (we'll only do this for spin-0 for simplicity
print("Sample covariance")
nsamp = 100
covar_sample = np.zeros([n_ell, n_ell])
mean_sample = np.zeros(n_ell)
for i in np.arange(nsamp):
    print(i)
    f, _ = get_sample_field()
    cl = compute_master(f, f, w00)[0]
    covar_sample += cl[None, :]*cl[:, None]
    mean_sample += cl
mean_sample /= nsamp
covar_sample = covar_sample / nsamp
covar_sample -= mean_sample[None, :] * mean_sample[:, None]

# Let's plot them:
plt.figure()
plt.imshow(covar_TT_TT, origin='lower', interpolation='nearest')
plt.figure()
plt.imshow(covar_sample, origin='lower', interpolation='nearest')
plt.figure()
plt.imshow(covar_TT_TT-covar_sample, origin='lower', interpolation='nearest')
plt.show()
```

CHAPTER 10

Example 9: Rectangular pixels

This sample script showcases the use of the NmtField class for maps with rectangular pixellization.

```
import matplotlib.pyplot as plt
import pymaster as nmt
from astropy.io import fits
from astropy.wcs import WCS

# This script showcases the use of the NaMaster to compute power spectra
# for curved-sky fields with rectangular pixelization.
#
# Note that NaMaster does not support any kind of rectangular pixellization.
# The specific kind supported is pixels defined using the CAR (Plate-Carree)
# projection and with Clenshaw-Curtis weights (i.e. the WCS reference pixel
# must lie on the equator, and the full latitude range must be divided
# exactly into pixels, with one pixel centre at both poles.

# Fields with rectangular pixelization are created from a WCS object that
# defines the geometry of the map.
hdul = fits.open("benchmarks/msk_car.fits")
wcs = WCS(hdul[0].header)
hdul.close()

# Read input maps
# a) Read mask
mask = fits.open("benchmarks/msk_car.fits")[0].data
# b) Read maps
mp_t, mp_q, mp_u = fits.open("benchmarks/mps_car.fits")[0].data
# You can also read and use contaminant maps in the same fashion.
# We'll skip that step here.

# # # #  Create fields
# Create spin-0 field. Pass a WCS structure do define the rectangular pixels.
f0 = nmt.NmtField(mask, [mp_t], wcs=wcs, n_iter=0)
# Create spin-2 field
```

(continues on next page)

(continued from previous page)

```
f2 = nmt.NmtField(mask, [mp_q, mp_u], wcs=wcs, n_iter=0)

# Let's check out the maps.
# First the original map
plt.figure()
plt.title("Original map")
plt.imshow(mp_t, interpolation='nearest', origin='lower')
# Now the map processed after creating the NmtField. Note that `get_maps()`
# will return flattened maps, so you need to undo that.
plt.figure()
plt.title("Map from NmtField")
plt.imshow(f0.get_maps().reshape([mp_t.shape[0], -1]),
           interpolation='nearest', origin='lower')
# You'll notice that, after creating the NmtField, the maps get extended
# to cover the full 2*pi azimuth range. If you want to recover the original
# map, you'll need to cut that out.
plt.figure()
plt.title("Map from NmtField, cut")
plt.imshow(f0.get_maps().reshape([mp_t.shape[0], -1])[:, :mp_t.shape[1]],
           interpolation='nearest', origin='lower')
plt.show()
```

CHAPTER 11

Indices and tables

- genindex
- modindex
- search

Python Module Index

p

Index