emuPK

Release 0.0.1

Arrykrishna Mootoovaloo

Aug 02, 2022

CONFIGURATIONS

Emulator for 3D matter power spectrum

`emuPK` is an emulator for generating the 3D matter power spectrum which can be used in conjunction with a weak lensing likelihood code to derive constraints on cosmological parameters. It is built based on the following parameters and prior range:

Table 1: Definition of the parameter inputs to the emulator

Parameters	Description	Prior
$\Omega_{\rm cdm}h^2$	CDM density	$\mathcal{U}[0.06, 0.40]$
$\Omega_{\rm b}h^2$	Baryon density	$\mathcal{U}[0.019, 0.026]$
$\ln(10^{10}A_s)$	Scalar spectrum amplitude	$\mathcal{U}[1.7, 5.0]$
n_s	Scalar spectral index	$\mathcal{U}[0.7, 1.3]$
h	Hubble parameter	$\mathcal{U}[0.64, 0.82]$
Σm_ν	Neutrino mass (eV)	$\mathcal{U}[0.06, 1.0]$

Including the neutrino mass is an optional step in the process of building the emulator. For further details, please refer to the explanation below.

Citation

If you use this code in your research, please cite these papers 2005.06551 and 2105.02256:

```
Mootoovaloo, A., Heavens, A. F., Jaffe, A. H., & Leclercq, F., 'Parameter
inference for weak lensing using Gaussian Processes and MOPED' MNRAS, 497,
2213 (2020)
```

```
Mootoovaloo, A., Jaffe, A. H., Heavens, A. F., & Leclercq, F., 'Kernel-based
emulator for the 3D matter power spectrum from CLASS', A&C, 38, 100508
(2022).
```

We explain briefly how the emulator can be used in a weak lensing analysis and we also provide an example to illustrate the performance of the emulator. Here we provide a brief overview of the code structure. Please see Github for full code structure. We do not cover each code in full detail here.

CHAPTER

ONE

BASIC SETUP

There are multiple possibilities which we consider when building the emulator. All the configurations are provided in a setting file here. In particular, one can emulate the 3D matter power spectrum directly or one can decompose it into three components as:

$P_\delta(k, z) = A(z)[1 + q(k, z)]P_{\rm lin}(k, z_0)$

If we choose to do the latter, then this can be specified using

```
components = True
```

Moreover, one can choose to include the neutrino mass in the emulating scheme. If one opt not to do so, the neutrino mass can be fixed to a value. In many weak lensing analysis, this mass is fixed to 0.06 eV.

```
neutrino = False

if not neutrino:
    fixed_nm = {'M_tot': 0.06}
```

Once the LHS points are generated and scaled according to the range of cosmological values we want, we found that CLASS gets permanently stuck at some of these points. Hence, to acoid this, we allocate a timeslot to allow CLASS to run (below, we have fixed it to 60 seconds). On average, the time taken for a single forward simulation is around 30 seconds, hence can be used as a guide to fix the value below.

```
timeout = 60
```

Next, we consider the lower and upper limits of redshift and wavenumber. The minimum and maximum redshift can be set using the variables `zmin` and `zmax` respectively. However, for k, for accurate power spectrum calculation, $k_{\rm max}$, `k_max_h_by_Mpc` below is set to a very high value (private communication with CLASS developers). However, when calculating the power spectrum, we define another variable, `kmax` (which is less than the other one) to set the upper limit of k.

```
# minimum redshift
zmin = 0.0

# maximum redshift
zmax = 4.66

# for accurate power spectrum, set to a very high value, for example, 5000
k_max_h_by_Mpc = 5000.

# our wanted kmax
kmax = 50.0
```

(continues on next page)

(continued from previous page)

```
# minimum of k
k_min_h_by_Mpc = 5E-4
```

CHAPTER

TWO

EXAMPLE

A notebook is available on Github to illustrate the different test cases. Below we show an example of the different reconstructed power spectra (comparing the power spectra generated using the emulator and the solver, CLASS). In particular, we have the

- linear matter power spectrum at the reference redshift
- the non-linear matter power spectrum at a fixed redshift
- the non-linear matter power spectrum, with baryon feedback at a fixed redshift

and we can also use a specific redshift distribution to compute the auto- and cross- weak lensing power spectra.

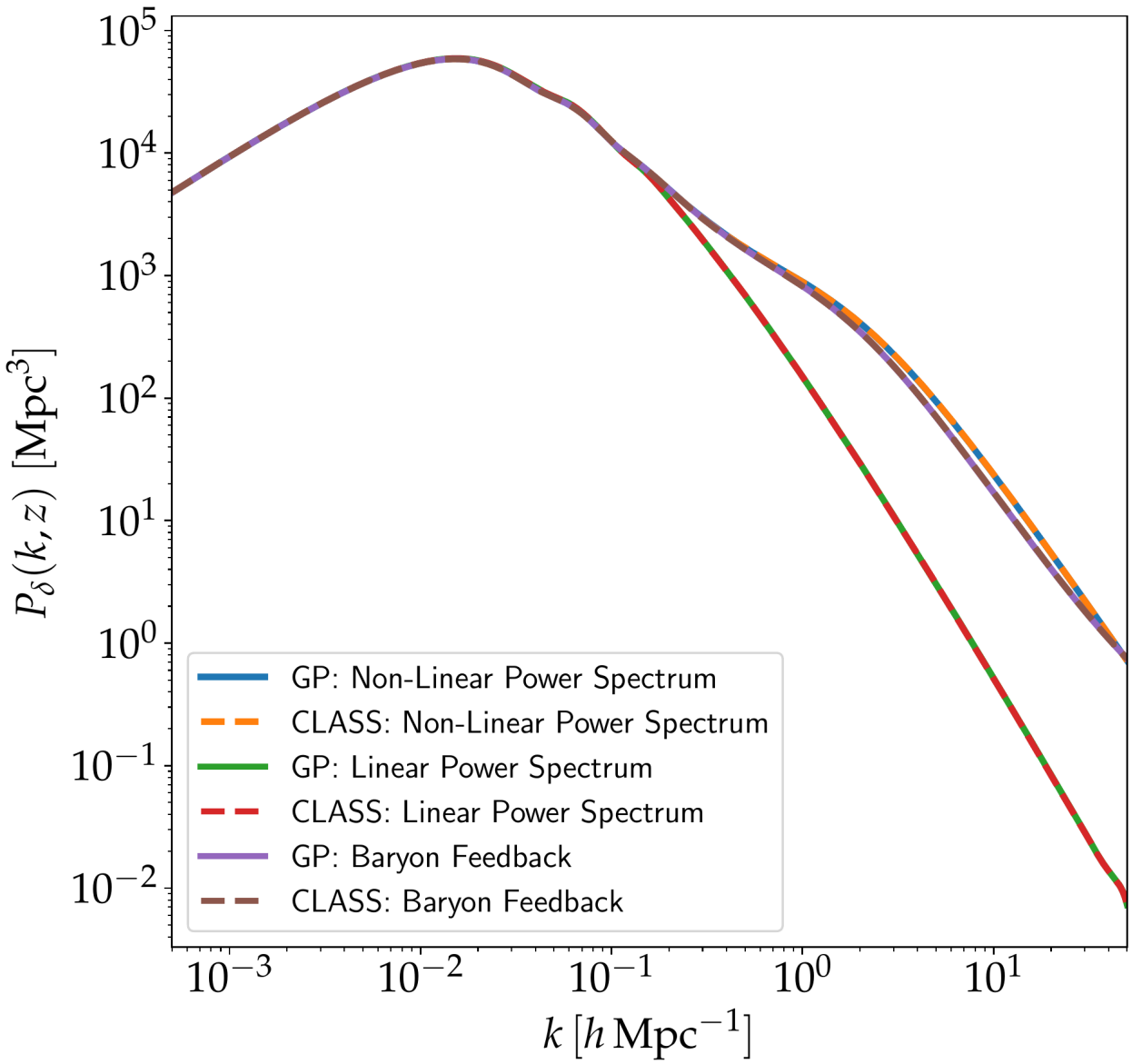

Fig. 1: Comparison between the emulator and CLASS at fixed redshift and input parameters.

CHAPTER

THREE

EMULATOR BUILD-UP

Building the emulator requires different stages:

- generating the training points
- training the Gaussian Processes

The inputs are generated using Latin Hypercube sampling and these are scaled using the `priors` module below. One can specify a prior using the dictionary format as follows:

```
{'distribution': 'uniform', 'specs': [0.06, 0.34]}
```

where `specs` are the specificifications for the distribution. In the example above, 0.06 is the minimum and the maximum is $0.06 + 0.34 = 0.40$. Note that we are using `scipy.stats` and hence the above convention. The GP models are trained in parallel (number of cores available on your computer) and are stored.

3.1 Predictions

Calculate and store the predictions at test points in parameter space

`predictions.`**`main`**(*ntest: int = 1*, *a_bary: float = 1.0*) → Tuple[numpy.ndarray, numpy.ndarray]

Calculate the non-linear matter power spectrum at test points in parameter space

Param ntest (int) - the number of test points we want to use

Param a_bary (float) - the Baryon Feedback parameter (default: 1.0)

Returns pk_class (np.ndarray) - the 3D power spectrum calculated by CLASS

Returns pk_gp (np.ndarray) - the 3D power spectrum calculated by the emulator

3.2 Priors

Module for important calculations involving the prior. For example,

- when scaling the Latin Hypercube samples to the appropriate prior range
- when calculating the posterior if the emulator is connected with an MCMC sampler

`priors.`**`all_entities`**(*dict_params*)

Generate all the priors once we have specified them.

Param dict_params (dict) - a list containing the description for each parameter and each description (dictionary) contains the following information:

- distribution, specified by the key 'distribution'
- parameter name, specified by the key 'parameter'
- specifications, specified by the key 'specs'

Returns

record (list) - a list containing the prior for each parameter, that is, each element contains the following information:

- parameter name, specified by the key 'parameter'
- distribution, specified by the key 'distribution'

`priors.`**`entity`**(*dictionary*)

Generates the entity of each parameter by using scipy.stats function.

Param dictionary (dict) - a dictionary containing information for each parameter, that is,

- distribution, specified by the key 'distribution'
- specifications, specified by the key 'specs'

Returns dist (dict) - the distribution generated using scipy

`priors.`**`log_prod_pdf`**(*desc: dict*, *parameters: dict*) → float

Calculate the log-product for a set of parameters given the priors

Param desc (dict) - dictionary of parameters

Param parameters (np.ndarray) - an array of parameters

Returns log_sum (float) - the log-product of when the pdf of each parameter is multiplied with another

3.3 Training

Routine to train all GPs in parallel to emulate the 3D Power Spectrum

`training.`**`main`**(*directory: str = 'semigps'*) → None

Main function to train all the Gaussian Process models.

Param directory (str) - directory where the GPs are stored.

Returns None

`training.`**`parallel_training`**(*arguments: list*) → None

Call the parallel processing routine here

Param arguments (list) - list of arguments (inputs) to train the GPs

Returns None

`training.`**`train`**(*cosmologies: numpy.ndarray*, *target: numpy.ndarray*, *folder_name: str*, *fname: str*, *kwargs: dict*) → None

Function to train GPs

Param cosmologies (np.ndarray) : array of size N_train x N_dim for the inputs to the GP

Param target (np.ndarray) : an array for the targets (function)

Param folder_name (str) : name of the folder where the outputs are stored

Param fname (str) : name of the GP output

Param kwargs (dict) : a dictionary with the settings for the GPs, for example, lambda_cap = 1000

`training.worker`(*args: list*) → None

The argument here is simply the name of the output vector

Param args (list) : list containing the arguments to be fed for training GPs

3.4 Training Points

Routine to scale the Latin Hypercube samples according to the prior and evaluate the power spctrum at these points.

`trainingpoints.CLASS_RUN`(*module: object*, *parameter: numpy.ndarray*, *index: int*) → Tuple[bool, dict]

Run CLASS given an input parameter to generate the training points (outputs)

Param module (object) - the CLASS module

Param parameter (np.ndarray) - the input cosmology, either 5 dimensions or 6 dimensions

Index i*th cosmology from the LHS file

Returns state (bool), quantities (dict) - state indicates if the run is successful, quantities contain the important quantities generated

class `trainingpoints.trainingset`(*lhs: str = 'maximin_1000_6D'*)

Bases: `object`

Runs CLASS at the LHS points generated using the maximin procedure. If we want to sample the neutrino mass, then, please use maximin_1000_6D as input (assuming it has already been generated), otherwise please use maximin_1000_5D.

scale(*save: bool = True*) → numpy.ndarray

Scale the LHS according to the prior range. See setting file to set up the priors for the LHS samples.

Param save (bool) - if True, the scaled inputs (cosmologies) will be written to a file

Returns cosmologies (np.ndarray) - the scaled inputs

targets(*cosmologies: numpy.ndarray*, *save: bool = False*) → numpy.ndarray

Generate the power spectrum at the specfic cosmologies

Param save (bool) - if True, the generated power spectrum will be saved in a directory. Note that the power spectrum is of shape (nk x nz), for example, 40 x 20. So the final shape will be of size (ncosmo x nk x nz). The power spectrum is flattened in this case, so we save a file of size 1000 x 800 (ncosmo = 1000, nk = 40, nz = 20). Therefore, we will have 800 separate GPs in this example.

Param cosmologies (np.ndarray) - set of cosmologies where we want to run CLASS

Param save (bool) - if True, the generated targets (training points/ power spectrum) will be saved in a directory

Returns components (dict) - a list of the different quantities (growth factor, linear matter power spectrum, q function) evaluated at different cosmologies or

Returns pk_non (np.ndarray) - the power spectrum evaluated at each cosmology

CHAPTER

FOUR

3D MATTER POWER SPECTRUM

The emulation procedure can be broken down into multiple different steps. The goal is to generate both the 3D matter power spectrum and the different weak lensing power spectra. Hence we have multiple layers, which incvolve generating the important quantities from CLASS (the forward simulations) and we also require the redshift distribution. To be more generic, we also allow the calculation of the 3D matter power spectrum using CLASS as an option.

4.1 Cosmology Functions

Some important functions for power spectrum and likelihood calculation

`cosmofuncs.`**`bar_fed`**(*k*, *z*, *a_bary=0.0*)

Fitting formula for baryon feedback following equation 10 and Table 2 from J. Harnois-Deraps et al. 2014 (arXiv.1407.4301)

Param k (np.ndarray): the wavevector

Param z (np.ndarray): the redshift

Param A_bary (float): the free amplitude for baryon feedback (Default: 0.0)

Returns b^2(k,z): bias squared

`cosmofuncs.`**`cosmo_params`**(*d: dict*) → dict

Given a dictionary for all the parameters, this function returns a dictionary only for the inputs to the emulator

Param d (dict) - a dictionary with all the parameters (keys and values)

Returns emu_param (dict) - a dictionary with inputs to the emulator

`cosmofuncs.`**`delete_module`**(*module*)

Delete Class module - accumulates memory unnecessarily

Param module (classy.Class) - the Class module

`cosmofuncs.`**`dictionary_params`**(*d: dict*) → Tuple[dict, dict, dict]

A dictionary for storing all the parameters

The parameters are organised in the following order (using CLASS and MontePython notations):

- omega_cdm
- omega_b
- ln10^{10}A_s
- n_s
- h

- M_tot

Param par (np.ndarray): parameters for the inference

Returns cosmo (dict) : a dictionary for the cosmology setup

Returns other (dict) : a dictionary for the neutrino settings

Returns neutrino (dict) : a dictionary for the neutrino

Returns nuisance (dict) : a dictionary which contains the baryon feedback parameter

`cosmofuncs.ds_ee` (*qs: list*, *quant: dict*) → numpy.ndarray

Calculates the double sum for the EE power spectrum

Param qs (list) - a list of the functions Q

Param quant (dict) - a dictionary containing all precomputed and basic quantities

Returns dsum (np.ndarray) - the double sum

`cosmofuncs.ds_gi` (*f_gi: numpy.ndarray*, *quant: dict*) → numpy.ndarray

Calculates the double sum for the II power spectrum. In this case, it is a single summation

Param f_ii (np.ndarray) - an array for the function F_II

Param quant (dict) - a dictionary containing all precomputed and basic quantities

Returns dsum (np.ndarray) - the double sum

`cosmofuncs.ds_ii` (*f_ii: numpy.ndarray*, *quant: dict*) → numpy.ndarray

Calculates the double sum for the II power spectrum. In this case, it is a single summation

Param f_ii (np.ndarray) - an array for the function F_II

Param quant (dict) - a dictionary containing all precomputed and basic quantities

Returns dsum (np.ndarray) - the double sum

`cosmofuncs.get_critical_density` (*small_h*)

The critical density of the Universe at redshift 0.

Param small_h (float) - the Hubble parameter

Returns rho_crit_0 (float) - the critical density at redshift zero

`cosmofuncs.get_factor_ia` (*quant: dict*, *redshift: numpy.ndarray*, *amplitude: float*, *exponent=0.0*) → numpy.ndarray

Calculates F(chi) - equation 23 in Kohlinger et al. 2017.

Param quant (dict) - a dictionary containingthe critical density, omega matter, linear groth rate, Hubble parameter

Param redshift (np.ndarray) - a vector for the redshift

Param amplitude (float) - the amplitude due to intrinsic alignment

Param exponent (float) - an exponential factor (default: 0.0) - not used in inference

`cosmofuncs.integration_q_ell` (*f_ell*, *chi*, *order*)

Calculate Q_ell for all possible pairs of chi

Param f_ell (np.ndarray) : array for F_ell(chi) - see notes for further details)

Param chi (np.ndarray) : array for the comoving radial distance

Param order (int) : either 0 or 1 or 2

Returns q_ell (np.ndarray) - array of size nells x nz (see notes for further details)

`cosmofuncs.marg_params`(*d: dict*)

Returns different dictionaries to be used at different parts of the likelihood code.

Param d (dict) - a dictionary with all the parameters (keys and values)

Returns different dictionaries (based on conditions in the setting file)

`cosmofuncs.mk_dict`(*l1: list*, *l2: list*)

Create a dictionary given a list of string and a list of numbers

Param l1 (list) - list of string (parameter names)

Param l2 (list) - list of values (values of each parameter)

Returns d (dict) - a dictionary consisting of the keys and values

`cosmofuncs.nuisance_params`(*d: dict*) → dict

Given a dictionary for all the parameters, this function returns a dictionary only for the inputs to the emulator

Param d (dict) - a dictionary with all the parameters (keys and values)

Returns emu_param (dict) - a dictionary with inputs to the emulator

`cosmofuncs.ps_ee`(*index_i: int*, *index_j: int*, *heights: dict*, *dsum: numpy.ndarray*) → numpy.ndarray

Calculates the weak lensing power spectrum using the double sum approach

Param index_i (int) - the ith tomographic bin

Param index_j (int) - th j^th tomographic bin

Param heights (dict) - a dictionary with keys: h0, h1, h2

Param dsum (np.ndarray) - part of the double sum

Returns wl_ee (np.ndarray)- the EE weak lensing power spectrum

`cosmofuncs.ps_gi`(*index_i: int*, *index_j: int*, *heights: dict*, *dsum: numpy.ndarray*) → numpy.ndarray

Calculates the weak lensing power spectrum using the double sum approach

Param index_i (int) - the ith tomographic bin

Param index_j (int) - th j^th tomographic bin

Param heights (dict) - a dictionary with keys: h0, h1, h2

Param dsum (np.ndarray) - part of the double sum

Returns wl_gi (np.ndarray)- the II weak lensing power spectrum

`cosmofuncs.ps_ii`(*index_i: int*, *index_j: int*, *heights: dict*, *dsum: numpy.ndarray*) → numpy.ndarray

Calculates the weak lensing power spectrum using the double sum approach

Param index_i (int) - the ith tomographic bin

Param index_j (int) - th j^th tomographic bin

Param heights (dict) - a dictionary with keys: h0, h1, h2

Param dsum (np.ndarray) - part of the double sum

Returns wl_ii (np.ndarray)- the II weak lensing power spectrum

`cosmofuncs.runTime`(*target: object*, *func: object*, *param: dict*, *timeout: int = 60*)

Execute the above function(s) within the allocated time frame

Param target (object) - the target we want to time

Param func (object) - the module for calculating the non-linear matter power spectrum

Param param (dict) - a dictionary of input cosmology

Param timeout (int) - the alloated time window to allow the code to run

Returns state (bool) - True if CLASS runs successfully

Returns results (dict) - if CLASS runs successfully

`cosmofuncs.timeOut` (*func: object*, *param: dict*, *q: object*) → None

Calculate a (general) function within an allocated time frame

Param func (object) - the module for calculating the non-linear matter power spectrum

Param param (dict) - a dictionary of input cosmology

Param q (object) - the multiprocessing queue

`cosmofuncs.timeOutComponents` (*func: object*, *param: dict*, *q: object*) → None

Calculate the components of the non-linear matter power spectrum

Param func (object) - the module for calculating the non-linear matter power spectrum

Param param (dict) - a dictionary of input cosmology

Param q (object) - the multiprocessing queue

4.2 Redshift

Setup for the redshift distributions

class `redshift.nz_dist` (*zmin: float = None*, *zmax: float = None*, *nzmax: int = None*)

Bases: `object`

nz_gaussian (*z0: float*, *sigma: float*) → numpy.ndarray

Gaussian n(z) distribution for the tomographic bin

$n(z) = \frac{1}{2\pi\sigma}\exp(-\frac{1}{2}\frac{(z-z_0)^2}{\sigma^2})$

nz_model_1 (*zm: float*) → numpy.ndarray

Calculate the analytic function

$n(z) = z^2\exp(-\frac{z}{z_0})$

nz_model_2 (*z0: float*, *alpha: float = 2*, *beta: float = 1.5*)

https://arxiv.org/pdf/1502.05872.pdf

Calculate the analytic function

$n(z) = z^\alpha\exp(-(\frac{z}{z_0})^\beta)$

4.3 Power Spectrum Calculation

Module to generate the matter power spectrum either from CLASS or using the emulator

class spectrumcalc.**matterspectrum**(*emulator=True*)

Bases: cosmology.spectrumclass.powerclass

Routine to sample the cosmological and nuisance parameters. We have variaous options here. We can either use the emulator to calculate the 3D matter power spectrum or we can use CLASS itself.

gp_gradient(*testpoint: numpy.ndarray*, *order: int = 1*) → numpy.ndarray

Calculate the gradient of the power spectrum

Param testpoint (np.ndarray) : a testpoint

Param order (int) : 1 or 2 (1 refers to first derivative and 2 refers to second)

Returns the derivatives of the power spectrum (of size (nk x nz) x ndim), for example 800 x 7

int_grad_pk_nl(*params: dict*, *order: int = 1*, *int_type: str = 'cubic'*, *eps: list = [0.001]*, ***kwargs*) → dict

Calculates the gradient of the power spectrum at a point in parameter space

Param params (dict) : a point within the prior box

Param eps (float) : epsilon - using central finite difference method to calculate gradient

Param int_type (str) : type of interpolation (linear, cubic, quintic). Default is cubic

Returns grad (dict) : a dictionary containing the gradient of each parameter

int_pk_nl(*params: dict*, *a_bary: float = 0*, *int_type: str = 'cubic'*, ***kwargs*) → numpy.ndarray

Calculate the (interpolated) 3D matter power spectrum at a test point

Param parameters (dict) : a dictionary of the test point in parameter space

Param a_bary (float) : the baryon feedback parameter (DEFAULT = 0)

Param int_type (str) : type of interpolation (linear, cubic, quintic). Default is cubic

Returns spectrum (np.ndarray) : the interpolated 3D matter power spectrum

load_gps(*directory: str = 'semigps'*) → list

Load all the trained Gaussian Processes.

Param directory (str) - the directory where the GPs are stored (depends on our choice:

- zero mean GP
- semi-GP (DEFAULT: semigps)

Returns gps (list) - a list containing all the GPs

mean_prediction(*parameters: numpy.ndarray*) → numpy.ndarray

Calculate the mean prediction from all the GPs on the grid k x z

Param parameters (np.ndarray) - a vector of the test point in parameter space

Returns pred (np.ndarray) - the predictions from all the GPs

pk_nl_pred(*params: dict*) → numpy.ndarray

Calculate the non linear matter power spectrum at a given test point

Param parameters (np.ndarray) - a vector of the test point in parameter space

Returns pk (np.ndarray) - the non linear matter power spectrum (reshaped in nk x nz)

4.4 CLASS Power Spectrum

Calculate the matter power spectrum using CLASS

class spectrumclass.**powerclass**

Bases: object

Uses CLASS to compute the matter power spectrum

class_compute(*parameters: dict*)

Calculate the relevant quantities using CLASS

Param parameters (dict) : dictionary of input parameters to CLASS

Returns class_module : the whole CLASS module (which contains distances, age, temperature and others)

configurations() → None

Calculate and store the basic configurations which will be used by CLASS. This requires setting up a dictionary for the quantities we want CLASS to take as default and also the quantity we want CLASS to output.

derivatives(*params: dict*, *order: int = 1*, *eps: list = [0.001]*, ***kwargs*) → numpy.ndarray

Calculates the gradient of the power spectrum at a point in parameter space

Param params (dict) : a point within the prior box

Param eps (float) : epsilon - using central finite difference method to calculate gradient

Returns grad (list) : an array containing the gradient of each parameter (of size (nk x nz) x ndim), for example 800 x 7

pk_nonlinear(*parameters: dict*, ***kwargs*) → numpy.ndarray

Calculate the 3D matter power spectrum based on the emulator setting file

Param parameters (dict) - inputs to calculate the matter power spectrum

Returns pk_matter (np.ndarray) - the 3D matter power spectrum

pk_nonlinear_components(*parameters: dict*, *zref: float = 0.0*, ***kwargs*) → dict

The non-linear 3D matter power spectrum can be decomposed into:

P_nonlinear (k,z) = A(z) [1 + q(k,z)] P_linear (k,z0)

Calculates the following quantities:

- non linear 3D matter power spectrum, P_nonlinear(k,z)
- linear matter power spectrum (at the reference redshift - see setting file)
- the growth factor, A(z)
- the quantity q(k,z)

Param parameters (dict) - inputs to calculate the matter power spectrum

Param zref (float) - the reference redshift at which the linear matter power spectrum is calculated (DEFAULT: 0.0)

Returns quantities (dictionary) - dictionary consisting of the nonlinear power spectrum, linear power spectrum, growth factor and the non-linear function q(k,z)

pk_nonlinear_timed(*parameters: dict*) → dict

Calculate the non linear matter power spectrum but within an allocated period of time

Param parameters (dict) - a dictionary for inputs to CLASS

Returns quant (dict) - a dictionary of the calculated quantities

4.5 Weak Lensing Spectra

Calculate the Weak Lening Power spectra using simulator/emulator

class `weaklensing.spectra`(*emu: bool = False*, *dir_gp: str = 'semigps'*)

Bases: `cosmology.spectrumcalc.matterspectrum`, `cosmology.redshift.nz_dist`

basic_class(*cosmology: dict*) → dict

Calculates basic quantities using CLASS

Param d (dict) - a dictionary containing the cosmological and nuisance parameters

Returns quant (dict) - a dictionary with the basic quantities

n_of_z(*zcenter: list*, *model_name: str*, *dist_prop: dict*, *dist_range: dict = {}*) → dict

Calculate the (mid-) redshift and the heights

Param zcenter (list) - a list of the center of source distribution

Param nodel_name (str) - name of the n(z) we want to use - the following currently supported

1) model_1
2) model_2
3) gaussian

Param dist_range (dict) - a dictionary with the following key words: zmin, zmax, nzmax

Param dist_prop (dict) - a dictionary with the key words for the specific distribution,for example:

1) nz_model_2: dist_prop = {alpha: 2, beta: 1.5}
2) nz_gaussian: dist_prop = {sigma: [0.25, 0.25]}

If we are using 2 tomographic bins in the latter

Returns red_args (dict) - a dictionary with the redshift and heights

pk_matter(*cosmo: dict*, *a_bary: float = 0.0*) → Tuple[dict, dict]

Calculate the non-linear matter power spectrum

Param d (dict) - a dictionary with all the parameters (keys and values)

Returns pk_matter (np.ndarray), quant (dict) - an array for the non-linear matter power spectrum and a dictionary with the important quantities related to cosmology

wl_power_spec(*cosmo: dict*, *a_bary: float = 0.0*) → Tuple[dict, dict, dict]

Power spectrum calculation using the functional form of the n(z) distribution

Param d (dict) - a dictionary for the parameters

Returns cl_ee (dict) - the auto- and cross- EE power spectra

Returns cl_gi (dict) - the auto- and cross- GI power spectra

Returns cl_ii (dict) - the auto- and cross- II power spectra

CHAPTER

FIVE

GAUSSIAN PROCESS

Here, we provide multiple code (although not all of them are integrated in the full pipeline) to learn a function between the inputs and the outputs. In the simple case, one can use a Gaussian Linear Model or a zero mean Gaussian Process. Note that throughout, we are using the Radial Basis Function (RBF) kernel. We can also opt for applying $\log_{10}$ transformation to the output. For the inputs, the pre-whitening step is highly recommended.

5.1 Algebra

Important linear algebra operations for Gaussian Process

`algebra.`**`diagonal`**(*matrix: numpy.ndarray*) → bool

Check if a matrix is diagonal

Param matrix (np.ndarray) : matrix of size N x N

Returns cond (bool) : if diagonal, True

`algebra.`**`matrix_inverse`**(*matrix: numpy.ndarray*, *return_chol: bool = False*) → numpy.ndarray

Sometimes, we would need the matrix inverse as well

If we are dealing with diagonal matrix, inversion is simple

Param matrix (np.ndarray) : matrix of size N x N

Param return_chol (bool) : if True, the Cholesky factor will be returned

Returns dummy (np.ndarray) : matrix inverse

If we also want the Cholesky factor:

Returns chol_factor (np.ndarray) : the Cholesky factor

`algebra.`**`solve`**(*matrix: numpy.ndarray*, *b_vec: numpy.ndarray*, *return_chol: bool = False*) → numpy.ndarray

Given a matrix and a vector, this solves for x in the following:

Ax = b

If A is diagonal, the calculations are simpler (do not require any inversions)

Param matrix (np.ndarray) : 'A' matrix of size N x N

Param b_vec (np.ndarray) : 'b' vector of size N

Param return_chol (bool) : if True, the Cholesky factor will be retuned

Returns dummy (np.ndarray) : 'x' in the equation above

If we want the Cholesky factor:

Returns chol_factor (np.ndarray) : the Cholesky factor is returned

5.2 Gaussian Linear Model

Routine for polynomial regression (Gaussian Linear Model)

class gaussianlinear.**GLM**(*theta: numpy.ndarray*, *y: numpy.ndarray*, *order: int = 2*, *var: float = 1e-05*, *x_trans: bool = False*, *y_trans: bool = False*, *use_mean: bool = True*)

Bases: object

Gaussian Linear Model (GLM) class for polynomial regression

compute_basis(*test_point: numpy.ndarray = None*) → numpy.ndarray

Compute the input basis functions

Param test_point (np.ndarray: optional) : if a test point is provided, phi_star is calculated

Returns phi or phi_star (np.ndarray) : the basis functions

do_transformation() → None

Perform all transformations

evidence() → numpy.ndarray

Calculates the log-evidence of the model

Returns log_evidence (np.ndarray) : the log evidence of the model

inv_noise_cov() → numpy.ndarray

Calculate the inverse of the noise covariance matrix

Returns mat_inv (np.ndarray) : inverse of the noise covariance

inv_prior_cov() → numpy.ndarray

Calculate the inverse of the prior covariance matrix

mat_inv (np.ndarray) : inverse of the prior covariance matrix (parametric part)

noise_covariance() → None

Build the noise covariance matrix

posterior_coefficients() → Tuple[numpy.ndarray, numpy.ndarray]

Calculate the posterior coefficients

beta_bar (np.ndarray) : mean posterior

lambda_cap (np.ndarray) : covariance of the regression coefficients

prediction(*test_point: numpy.ndarray*) → Tuple[numpy.ndarray, numpy.ndarray]

Given a test point, the prediction (mean and variance) will be computed

Param test_point (np.ndarray) : vector of test point in parameter space

Returns post_mean (np.ndarray) : mean of the posterior

Returns post_var (np.ndarray) : variance of the posterior

regression_prior(*mean: numpy.ndarray = None*, *cov: numpy.ndarray = None*, *lambda_cap: float = 1*) → None

Specify the regression prior (mean and covariance)

Param mean (np.ndarray) : default zeros

Param cov (np.ndarray) : default identity matrix

Param lambda_cap (float) : width of the prior covariance matrix (default 1)

5.3 Kernel

Functions to calculate the kernel matrix - currently support the Radial Basis Function

`kernel.rbf`(*x_train: numpy.ndarray*, *x_test: numpy.ndarray = None*, *params: numpy.ndarray = None*) → numpy.ndarray

Implementation of the Radial Basis Function

Param x_train (np.ndarray) : a matrix of size N x d (N > d)

Param x_test (np.ndarray) : a matrix (or vector)

Param params (np.ndarray) : kernel hyperparameters (amplitude and lengthscale)

Returns kernel_matrix (np.ndarray) : the kernel matrix

If the x_test is not part of the training set, following Rasmussen et al. (2006) the following will be returned:

Returns kernel_s (np.ndarray) : a vector of size N

Returns kernel_ss (np.ndarray) : a scalar (1 x 1) array

`kernel.squared_distance`(*x1: numpy.ndarray*, *x2: numpy.ndarray*, *scale: numpy.ndarray*) → numpy.ndarray

Calculate the pairwise Euclidean distance between two input vectors (or matrix)

Param x1 (np.ndarray) : first vector (or matrix if we have more than 1 training point)

Param x2 (np.ndarray) : second vector (or matrix if we have more than 1 training point)

Param scale (np.ndarray) : the characteristic lengthscales for the kernel

Returns distance (np.ndarray) : pairwise Euclidean distance between the two vectors/matrix

5.4 Optimisation

Function to train a GP

`optimisation.maximise`(*x_train: numpy.ndarray*, *y_train: numpy.ndarray*, *y_trans: bool = True*, *lambda_cap: float = 1*) → object

Function for training one GP

Param x_train (np.ndarray) : the inputs to the GP

Param y_train (np.ndarray) : the output from the traning point

Param y_trans (bool) : option to transform the target

Param lambda_cap (float) : the prior width on the regression coefficients

Returns gp_module (class) : Python class with the trained GP

5.5 GP with Basis Functions

Learn a function by specifiying an explicit set of basis function and model the residuals by a kernel.

class semigp.**GP**(*theta: numpy.ndarray*, *y: numpy.ndarray*, *var: float = 1e-05*, *order: int = 2*, *x_trans: bool = False*, *y_trans: bool = False*, *jitter: float = 1e-10*, *use_mean: bool = False*)

Bases: object

Module to learn a function which maps the inputs to the output. There are various important aspects in having a semi-parameteric Gaussian Process model. The parameteric part here is a polynomial function. Only order = 1 and order = 2 are currently supported. In addition, we also use a pre-whitening step at the input level and the code also supports log_10 transformation for the targets.

Param theta (np.ndarray) : matrix of size ntrain x ndim

Param y (np.ndarray) : output/target

Param var (float or np.ndarray) : noise covariance matrix of size ntrain x ntrain

Param x_trans (bool) : if True, pre-whitening is applied

Param y_trans (bool) : if True, log of output is used

Param jitter (float) : a jitter term just to make sure all matrices are numerically stable

Param use_mean (bool) : if True, the outputs are centred on zero

compute_basis(*test_point: numpy.ndarray = None*) → numpy.ndarray

Compute the input basis functions

Param test_point (np.ndarray) : if a test point is provdied, phi_star is calculated

Returns phi or phi_star (np.ndarray) : the basis functions

delete_kernel() → None

Deletes the kernel matrix from the GP module

derivatives(*test_point: numpy.ndarray*, *order: int = 1*) → Tuple[numpy.ndarray, numpy.ndarray]

If we did some transformation on the ouputs, we need this function to calculate the 'exact' gradient

Param test_point (np.ndarray) : array of the test point

Param order (int) : 1 or 2, referrring to first and second derivatives respectively

Returns grad (np.ndarray) : first derivative with respect to the input parameters

Returns gradient_sec (np.ndarray) : second derivatives with respect to the input parameters, if specified

do_transformation() → None

Perform all transformations

evidence(*params: numpy.ndarray*) → Tuple[numpy.ndarray, numpy.ndarray]

Calculate the log-evidence of the GP

Param params (np.ndarray) : kernel hyperparameters

Returns neg_log_evidence (np.ndarray) : the negative log-marginal likelihood

Returns -gradient (np.ndarray) : the gradient with respect to the kernel hyperparameters

fit(*method: str = 'CG'*, *bounds: numpy.ndarray = None*, *options: dict = {'ftol': 1e-05}*, *n_restart: int = 2*) → numpy.ndarray

The kernel hyperparameters are learnt in this function.

Param method (str) : the choice of the optimizer:

https://docs.scipy.org/doc/scipy/reference/generated/scipy.optimize.minimize.html

Recommend L-BFGS-B algorithm

Param bounds (np.ndarray) : the prior on these hyperparameters

Param options (dictionary) : options for the L-BFGS-B optimizer. We have:

```
options={'disp': None,
        'maxcor': 10,
        'ftol': 2.220446049250313e-09,
        'gtol': 1e-05,
        'eps': 1e-08,
        'maxfun': 15000,
        'maxiter': 15000,
        'iprint': - 1,
        'maxls': 20,
        'finite_diff_rel_step': None}
```

Param n_restart (int) : number of times we want to restart the optimizer

Returns opt_params (np.ndarray) : array of the optimised kernel hyperparameters

grad_pre_computations(*test_point: numpy.ndarray, order: int = 1*) → Tuple[numpy.ndarray, numpy.ndarray]

Pre-compute some quantities prior to calculating the gradients

Param test_point (np.ndarray) : test point in parameter space

Param order (int) : order of differentiation (default: 1) - not to be confused with order of the polynomial

Returns gradients (tuple) : first and second derivatives (if order = 2)

inv_noise_cov() → numpy.ndarray

Calculate the inverse of the noise covariance matrix

Param mat_inv (np.ndarray) : inverse of the noise covariance

inv_prior_cov() → numpy.ndarray

Calculate the inverse of the prior covariance matrix

Returns mat_inv (np.ndarray) : inverse of the prior covariance matrix (parametric part)

noise_covariance() → numpy.ndarray

Build the noise covariance matrix

Returns the initial pre-defined noise variance (either float or matrix)

posterior() → Tuple[numpy.ndarray, numpy.ndarray]

Computes the posterior distribution of beta and f (latent variables)

Note: Optimise for the kernel parameters first

Param post_mean (np.ndarray) : mean posterior

Param a_inv_matrix (np.ndarray) : covariance of all latent parameters

Returns post_mean (np.ndarray) : mean of the regression coefficient and the residuals

Returns a_inv_matrix (np.ndarray) : the full covariance matrix of teh estimated parameters

pred_original_function(*test_point: numpy.ndarray*, *n_samples: int = None*) → numpy.ndarray

Calculates the original function if the log_10 transformation is used on the target.

Param test_point (np.ndarray) - the test point in parameter space

Param n_samples (int) - we can also generate samples of the function (assuming we have stored the Cholesky factor)

Returns y_samples (np.ndarray) - if n_samples is specified, samples will be returned

Returns y_original (np.ndarray) - the predicted function in the linear scale (original space) is returned

prediction(*test_point: numpy.ndarray*, *return_var: bool = False*) → Tuple[numpy.ndarray, numpy.ndarray]

Predicts the function at a test point in parameter space

Param test_point (np.ndarray) : test point in parameter space

Param return_var (bool) : if True, the predicted variance will be computed

Returns mean_pred (np.ndarray) : the mean of the GP

Returns var_pred (np.ndarray) : the variance of the GP (optional)

regression_prior(*mean: numpy.ndarray = None*, *cov: numpy.ndarray = None*, *lambda_cap: float = 1*) → None

Specify the regression prior (mean and covariance)

Param mean (np.ndarray) : default zeros

Param cov (np.ndarray) : default identity matrix

Param lambda_cap (float) : width of the prior covariance matrix (default 1)

5.6 Transformation

Functions to transform the inputs and outputs

class transformation.**transformation**(*theta: numpy.ndarray*, *y: numpy.ndarray*)

Bases: object

Module to perform all relevant transformation, for example, pre-whitening the inputs and logarithm (supports log10 transformation) for the outputs.

x_transform() → numpy.ndarray

Transform the inputs (pre-whitening step)

Returns theta_trans (np.ndarray) : transformed input parameters

x_transform_test(*xtest: numpy.ndarray*) → numpy.ndarray

Given a test point, we transform the test point in the appropriate basis

Param xtext (np.ndarray) : a vector of dimension d for the test point

Returns x_trans (np.ndarray) : the transformed input parameters

y_inv_transform_test(*y_test: numpy.ndarray*) → numpy.ndarray

Given a response (a prediction), this function will do the inverse transformation (from log_10 to the original function).

Param y_test (float or np.ndarray) : a test (transformed) response (output)

Returns y_inv (np.ndarray) : original (predicted) output

y_transform() → numpy.ndarray
Transform the output (depends on whether we want this criterion)

If all the outputs are positive, then y_min = 0, otherwise the minimum is computed and the outputs are shifted by this amount before the logarithm transformation is applied

Returns y_trans (np.ndarray) : array for the transformed output

y_transform_test(*y_original: numpy.ndarray*) → numpy.ndarray
Given a response/output which is not in the training set, this function will do the forward log_10 transformation.

Param y_original (float or np.ndarray) : original output

Returns y_trans_test (array) : transformed output

5.7 GP with Zero Mean

Zero Mean Gaussian Process

class zerogp.**GP**(*theta: numpy.ndarray*, *y: numpy.ndarray*, *var: float = 1e-05*, *x_trans: bool = False*, *y_trans: bool = False*, *use_mean: bool = False*)
Bases: object

Module to perform a zero mean Gaussian Process regression. One can also specify if we want to apply the pre-whitening step at the input level and the logarithm transformation at the output level.

Param theta (np.ndarray) : matrix of size ntrain x ndim

Param y (np.ndarray) : output/target

Param var (float or np.ndarray) : noise covariance matrix of size ntrain x ntrain

Param x_trans (bool) : if True, pre-whitening is applied

Param y_trans (bool) : if True, log of output is used

Param use_mean (bool) : if True, the outputs are centred on zero

delete_kernel() → None
Deletes the kernel matrix from the GP module

derivatives(*test_point: numpy.ndarray*, *order: int = 1*) → Tuple[numpy.ndarray, numpy.ndarray]
If we did some transformation on the ouputs, we need this function to calculate the 'exact' gradient

Param test_point (np.ndarray) : array of the test point

Param order (int) : 1 or 2, referrring to first and second derivatives respectively

Returns grad (np.ndarray) : first derivative with respect to the input parameters

Returns gradient_sec (np.ndarray) : second derivatives with respect to the input parameters, if specified

do_transformation() → None
Perform all transformations

evidence(*params: numpy.ndarray*) → Tuple[numpy.ndarray, numpy.ndarray]
Calculate the log-evidence of the GP and the gradient with respect to the kernel hyperparameters

Param params (np.ndarray) : kernel hyperparameters

Returns neg_log_evidence (np.ndarray) : the negative log-marginal likelihood

Returns -gradient (np.ndarray) : the gradient with respect to the kernel hyperparameters

fit (*method: str = 'CG', bounds: numpy.ndarray = None, options: dict = {'ftol': 1e-05}, n_restart: int = 2*) → numpy.ndarray

The kernel hyperparameters are learnt in this function.

Param method (str) : the choice of the optimizer:

https://docs.scipy.org/doc/scipy/reference/generated/scipy.optimize.minimize.html

Recommend L-BFGS-B algorithm

Param bounds (np.ndarray) : the prior on these hyperparameters

Param options (dictionary) : options for the L-BFGS-B optimizer. We have:

```
options={'disp': None,
        'maxcor': 10,
        'ftol': 2.220446049250313e-09,
        'gtol': 1e-05,
        'eps': 1e-08,
        'maxfun': 15000,
        'maxiter': 15000,
        'iprint': - 1,
        'maxls': 20,
        'finite_diff_rel_step': None}
```

Param n_restart (int) : number of times we want to restart the optimizer

Returns opt_params (np.ndarray) : array of the optimised kernel hyperparameters

grad_pre_computations (*test_point: numpy.ndarray, order: int = 1*) → Tuple[numpy.ndarray, numpy.ndarray]

Pre-compute some quantities prior to calculating the gradients

Param test_point (np.ndarray) : test point in parameter space

Param order (int) : order of differentiation (default: 1) - not to be confused with order of the polynomial

Returns gradients (tuple) : first and second derivatives (if order = 2)

noise_covariance () → numpy.ndarray

Build the noise covariance matrix

Returns the pre-defined (co-)variance in its appropriate form

pred_original_function (*test_point: numpy.ndarray, n_samples: int = None*) → numpy.ndarray

Calculates the original function if the log_10 transformation is used on the target.

Param test_point (np.ndarray) - the test point in parameter space

Param n_samples (int) - we can also generate samples of the function, assuming we have stored the Cholesky factor

Returns y_samples (np.ndarray) - if n_samples is specified, samples will be returned

Returns y_original (np.ndarray) - the predicted function in the linear scale (original space) is returned

prediction (*test_point: numpy.ndarray, return_var: bool = False*) → Tuple[numpy.ndarray, numpy.ndarray]

Predicts the function at a test point in parameter space

Param test_point (np.ndarray) : test point in parameter space

Param return_var (bool) : if True, the predicted variance will be computed

Returns mean_pred (np.ndarray) : the mean of the GP

Returns var_pred (np.ndarray) : the variance of the GP (optional)

CHAPTER

SIX

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

INDEX

M

N

O

P

R

S

T

W

X

Y

Z