EuclidEmulator

Mischa Knabenhans
PhD Program in Astrophysics and Cosmology
Center for Theoretical Astrophysics and Cosmology (CTAC)
Institute for Computational Science (ICS)
University of Zurich
Zurich, Switzerland
mischak@physik.uzh.ch

August 22, 2018

Contents

1 From download to installation **2**
1.1 Prerequesites . . . 2
1.2 Dependencies . . . 2
1.2.1 REQUIRED LIBRARIES . . . 2
1.2.2 RECOMMENDED (BUT OPTIONAL) LIBRARIES . . . 2
1.3 Environment . . . 3
1.4 Download . . . 3
1.5 Building and Installing the Code . . . 4
1.5.1 Installing the python2 package *e2py* . . . 4
1.5.2 Installing the command line interface (CLI) only . . . 5

2 Usage **6**
2.1 General . . . 6
2.1.1 Parameter ranges . . . 6
2.1.2 Units . . . 6
2.2 Python wrapper *e2py* . . . 7
2.3 Command line interface (CLI) . . . 9

3 Code structure **12**
3.1 Overview . . . 12
3.2 Background information . . . 12
3.2.1 EuclidEmulator is based on pre-evaluated cosmological simulations . 12
3.2.2 Data post-processing . . . 12
3.3 libeuc: the actual emulator . . . 13
3.4 CLI: the command line interface . . . 14
3.5 wrapper: the python wrapper . . . 15
3.5.1 The Cython layer in 'EuclidEmulator_BackEnd.pyx' . . . 15
3.5.2 The e2py python package . . . 16

3.6 ClassPatch 25
3.7 examples 25

1 From download to installation

1.1 Prerequesites

The EuclidEmulatorruns on Linux and MacOS. It has been tested with the following operating systems

- Scientific Linux release 6.9 (on zBox4 supercomputer)
- SUSE Linux release 12.2 (on PizDaint supercomputer)
- Linux openSUSE LEAP 15.0
- macOS Sierra (10.12.6)
- macOS High Sierra (10.13.6)

1.2 Dependencies

The code depends on a couple of third party libraries. Some of them are needed in all cases (termed "REQUIRED"), some are optional in the sense that you can still use EuclidEmulator if they are not installed on your machine, but the functionality is severely reduced (basically to the core functionality of emulating the dark matter boost factor). We **highly recommend** to install all dependencies as only then the full power of EuclidEmulator can be exploited.

1.2.1 REQUIRED LIBRARIES

You need to have installed:

- GSL (see GNU Scientific Library webpage), version 1.3 or higher
- CMake (see CMake webpage), version 3.5 or higher

Both packages should be available via the package managers APT (Linux) and Homebrew (MacOS).

1.2.2 RECOMMENDED (BUT OPTIONAL) LIBRARIES

If you intend to build the python2 package, you further need the following set of third party libraries:

- Cython (see Cython webpage), version 0.27.3 or higher
- Numpy (tested version: 1.14.2)
- Scipy (tested version: 1.0.1)
- Pandas (tested version: 0.20.3)

- Contextlib (tested version: 0.5.5)

Notice that both older and newer versions of the listed third party library might also work. If you find a required library that is missing in this list or if you successfully test this piece of software with some older version of a required library, please let me know such that I can update the documentation.

1.3 Environment

If you want to install EuclidEmulatoron a machine, you must have write permission.

- **Yes, I do have root access on my machine:** Awesome, everything should work out-of-the-box. In case it does not, please inform me about it with detailed descriptions of your machine, hardware, operating system etc.
- **No, I do not have root access on my machine:** We recommend you to create a virtual environment (as this is a standard procedure in this situation). For instructions on how to do this we point you to this tutorial from "The Hitchhiker's Guide to Python". Scroll down to the section "Lower level: virtualenv" and follow the step-by-step instructions.

Once you have created your virtual environment, enter it and activate it. Now you are all set to continue…

1.4 Download

There are multiple ways to download EuclidEmulator.

1. **Clone the GitHub repository:** This approach is recommended if you want to stay up-to-date with the developments of EuclidEmulator from our side or if you want to contribute to it.

   On the main page of the EuclidEmulator repository you can hit the green "Clone or download" button. Fire up a command line terminal and enter

   ```
   git clone https://github.com/miknab/EuclidEmulator
   ```

   or similar with `ssh` instead of `https` (however, if you want to use `ssh` you need to sign in to GitHub).

2. **Download a (g)zipped version of the repository:** In case a snap shot of the current version of EuclidEmulatoris enough for you and you do not intend to stay up-to-date with the developments nor do you intend to push contributions, you can also download the repository. Clicking on the green "Clone or download" button on the main page of the EuclidEmulator repository you will see drop-down menu with a "Download ZIP" button. Click on it and the download starts.

   Alternatively, you can download the master branch of the repository from your command line terminal via

   ```
   wget https://github.com/miknab/EuclidEmulator/archive/master.tar.gz
   ```

   In both cases, extract the downloaded archive and enter it to continue.

1.5 Building and Installing the Code

1.5.1 Installing the python2 package *e2py*

Once you have downloaded the code (and, if necessary, extracted the archive), enter the directory called `wrapper`.

Step 1: create an out-of-build tree Inside it, create a directory for an out-of-source build of the code:

```
mkdir build
```

Of course, you can name that directory whatever you like. Now enter this new "build" directory

```
cd build
```

Step 2: configure with CMake Next, perform the configuration using cmake:

```
cmake [-DCMAKE_INSTALL_PREFIX=path/to/installation/directory] ..
```

The definition of a install prefix is optional if you have root access to the machine. If this is not the case, you should set the install prefix to a directory for which you have write access (if you followed our suggestion of creating a virtual environment, see subsection 1.3, you should provide the path to this very environment). In any case, make sure that in `path/to/installation/directory` there are the subdirectories `bin`, `lib`, `include` and `share`.

Step 3: compile To compile the code, just type

```
make
```

Step 4: install To install it, type

```
make install
```

or on a machine on which you have root access

```
sudo make install
```

and enter your password. In case you have not configured the install prefix correctly in the configuration step, this step will not work and throw an error message like

```
CMake Error at /path/to/EuclidEmulator/libeuc/build/cmake\_ install.↩
    cmake:41 (file):
file INSTALL cannot copy file
"/path/to/EuclidEmulator/libeuc/build/libeucemu.a" to
"/usr/local/lib/libeucemu.a".
Call Stack (most recent call first):
cmake\_ install.cmake:82 (include)

make: *** [Makefile:118: install] Error 1
```

In this case just ‘cd‘ out of your build directory, delete it, create it again (otherwise cmake will remember your old but wrong configuration settings), enter it and follow the steps 2 to 4 again with the correct settings.

1.5.2 Installing the command line interface (CLI) only

Once you have downloaded the code (and, if necessary, extracted the archive), enter the directory called ‘CLI‘.

The first four steps of the building and installing procedure are 100

Step 5: Add installation folder to ‘$PATH‘ *If you have root access, this step is probably not necessary (but you can double-check to be on the safe side)*.

In order to make your life easy you may want to add the installation folder to your ‘$PATH‘ by e.g. modifying your bashrc or bash_ profile file accordingly. If you prefer not to do so, remember that you’ll have to call the executable by providing the path every time, i.e. instead of just ‘ee‘ you would have to type ‘path/to/installation/directory/ee‘.

2 Usage

This user manual should provide a more in-depth explanation of how to use EuclidEmulator. If you are more interested in a quick start, please refer to the see [README]https://github.com/miknab/EuclidEmulator#quick-start).

2.1 General

EuclidEmulator is meant to be easy to use and to provide certain flexibility. Hence both, a python package and a command line interface are provided as the first one is very convenient to use while the CLI might be easier to include into bigger pipelines. Another reason to use the CLI could be that you prefer to work with CAMB instead of CLASS (the *e2py* package sofar only supports CLASS).

Below it is described in detail how to work with each of these tools.

The underlying model used to construct this emulator is the six parameter model consisting of the dark energy equation of state parameter w_0 and the five standard ΛCDM parameters being the baryonic energy density $\Omega_\mathrm{b}h^2 = \omega_\mathrm{b}$, the total matter energy density $\Omega_\mathrm{m}h^2 = \omega_\mathrm{m}$, the spectral index n_s, the (dimensionless) Hubble parameter h and the variance of the density fluctuation field σ_8 which for the power spectrum is a normalization constant.

2.1.1 Parameter ranges

Irrespective of whether you use EuclidEmulatorvia the python package *e2py* or the CLI, there is a fixed range for each cosmological parameter that has to be obeyed. For the emulation of the boost, the cosmological parameters have to be within the following ranges:

$$\begin{aligned} 0.0217 \leq \omega_\mathrm{b} &\leq 0.0233 \\ 0.1326 \leq \omega_\mathrm{m} &\leq 0.1526 \\ 0.9345 \leq n_\mathrm{s} &\leq 0.9965 \\ 0.6251 \leq h &\leq 0.7211 \\ -1.250 \leq w_0 &\leq -0.750 \\ 0.7684 \leq \sigma_8 &\leq 0.8614 \end{aligned} \tag{1}$$

The redshift has to be $0.0 \leq z \leq 5.0$.

Input values outside this range will produce an error. Notice that these parameter ranges do *not* define the same parameter space as the one used to construct the emulator (reported in Knabenhans et al. 2018) but a smaller one. The reason for this is that emulation near the boundaries of the construction parameter space might not lead to accurate results.

2.1.2 Units

The units of both ‘Plin and ‘Pnonlin‘ are given in (Mpc^3h^{-3}), the boost factor ‘B‘ is dimensionless and the wave numbers k are given in units of $h\ \mathrm{Mpc}^{-1}$.

2.2 Python wrapper *e2py*

For the description of how to use the python package *e2py* we assume that you followed the installation instruction for EuclidEmulatorhttps://github.com/miknab/EuclidEmulator/wiki/II)-From-download-to-installation. You can now use *e2py* and its functionality in a python interpreter or a jupyter notebook.

Getting started: Start by importing the *e2py* package as usual:

```
import e2py
```

Notice that this should work irrespective of your current working directory. In ipython, a jupyter notebook or most python IDEs you can now e.g. type 'e2py.' and hit tab. The submodules and available functions will be listed. You can get more detailed information about each item in the usual way of typing its name in the python prompt and append a question mark (?). For example

```
e2py.get\_boost?
```

will lead to the following output:

```
Signature:    e2py.get\_boost(emu\_pars\_dict, redshifts)
Docstring:
Signature:    get\_boost(emu\_pars\_dict, redshifts)

Description: Computes the non-linear boost factor for a cosmology
             defined in EmuParsArr (a numpy array containing the
             values for the 6 LCDM parameters) at specified
             redshift stored in a list or numpy.array.

Input types: python dictionary (with the six cosmological parameters)
             list or numpy.array

Output type: python dictionary

Related:     get\_plin, get\_pnonlin
File:        ~/anaconda2/lib/python2.7/site-packages/e2py/ee\_observables.p
Type:        function
```

All modules and functions will be explained in more detail in the chapter [code structure]https://github.com/miknab/EuclidEmulator/wiki/IV)-Code-Structure of this wiki.

Definition of a cosmology: All functions in *e2py* expect cosmologies to be defined in python dictionaries like so

```
MyCosmo = {'om\_b': 0.0219961, 'om\_m': 0.1431991, 'n\_s': 0.96, 'h': 0.67,
```

The keys have to be exactly the ones mentioned in this example. You can either define a cosmology this way manually or you can use the function 'read_parfile' in order to read in a parameter file (this allows to read in several cosmologies at once). As an example, the file 'example/TestParFile.csv' contains three cosmologies and looks as follows

```
om\_b        om\_m        n\_s     h          w\_0     sigma\_8
0.0219961    0.1431991    0.96     0.67      -1.00     0.83
0.02312      0.1501       0.956    0.72      -0.89     0.81
0.02173      0.1399       0.968    0.71      -0.95     0.80
```

The labels in the first line of this file will be used as labels in the resulting dictionaries. So, in principle any labels can be used and they can be used in any order. If you use a different set of variables, though, you are responsible yourself to convert the parameters such that they match the EUCLIDEMULATORparameters introduced above. Further, while by default the separator between labels and values is assumed to be ",", other seperators are allowed as well as is shown in this very example where a varying number of whitespaces is used as a separator. This 'TestParFile.csv' can be read in as follows:

```
e2py.read\_parfile('/path/to/TestParFile.csv', sep='\s+')
```

where we used a regular expression (regex) to specify the separator. As a result one obtains a list of three python dictionaries each containing one of the cosmologies specified in the parameter file (in order).

Core functionality: Computing boost factors and power spectra The main functionality of EUCLIDEMULATORversion 1.1 is the computation of dark matter power spectra $P(k)$. The full non-linear $P_{\rm nonlin}(k)$ is a product of the linear contribution $P_{\rm lin}(k)$ and the so called boost factor $B(k)$:

$$P_{\rm nonlin}(k) = P_{\rm lin}(k)B(k) \tag{2}$$

Each of these three functions can be accessed in EuclidEmulator. Assuming a cosmology is defined in a python dictionary called *MyCosmo*, the non-linear power spectrum can be computed via

```
z = 0.5 #setting the redshift to 0.5
Pnl = e2py.get\_pnonlin(MyCosmo, z)
```

'Pnl' is again a dictionary of the format ''Pnonlin': ..., 'Plin': ..., 'B': ..., 'k': ...' . Notice that the function `get_pnonlin` also allows for the redshift variable 'z' to be of type list or numpy.ndarrays. In this case the resulting 'Pnl' is a nested dictionary where each of the fields 'Pnonlin', 'Plin' and 'B' is a dictionary itself with the labels 'z1', 'z2', 'z3',... referring to the individual redshifts specified in the redshift variable.

The `get_boost` function is called in exactly the same way

```
z = 0.5 #setting the redshift to 0.5
Pnl = e2py.get\_boost(MyCosmo, z)
```

while for calling `get_plin` one has to provide in addition a vector of k values at which CLASS/classee evaluates the linear power spectrum.

What happens behind the scenes when calling `get_pnonlin`: When you compute the non-linear power spectrum with EuclidEmulator, first the `get_boost` is run which calls the actual wrapper of the core of EuclidEmulator, a static C library called 'libeuc.a. A dictionary containing a set of k values and the corresponding boost values is returned. *THE VECTOR WITH k VALUES RETURNED BY `get_boost` IS ALWAYS THE SAME.*

With the same cosmology and the additional information at what k values the boost factor was computed, `get_plin` is called for the same redshift values as `get_pnonlin`. The linear power spectrum is computed with classee, a patched version of the official classy which is the python wrapper of the CLASS code. For more information about ClassPatch and classee, please see the [GitHub page of ClassPatch]https://github.com/miknab/ClassPatch. The resulting linear power spectrum and the boost factor are multiplied to get the non-linear power spectra. Notice that all interpolation is done inside classee.

NOTICE: Even without classy you can still use EuclidEmulatorto emulate boost factors. You won't be able to compute full power spectra, though.

Further functionality There is further functionality "hidden" in the EuclidEmulatorcode. Apart from the function 'get_pconv' in the 'ee_observables' module (which is still disabled), this further functionality is in the module '_ee_lens' or or in '_internal'. Notice that we follow the common pythonic naming scheme by prepending these module names with a "_" to make clear that they should be considered "private". The reason for this is that these modules contain functions that are not yet tested enough.

If you manually import and use any module whose name starts with a "_", we do not take any responsibility for the outcome.

Many functions will become "public" in the future as soon as they have undergone enough testing.

Auxiliary functions: There is a hand full of auxiliary functions available in *e2py*. What they do in detail and how they have to be called is described in their docstrings and in the [code structure]https://github.com/miknab/EuclidEmulator/wiki/IV)-Code-Structure part of this wiki.

2.3 Command line interface (CLI)

General usage of the CLI: An executable called "ee" will be created. To run the executable type

$$\text{ee } < \omega_\text{b} > < \omega_\text{m} > < \text{n}_\text{s} > < \text{h} > < \text{w}_0 > < \sigma_8 > < \text{z}_1 > \ [< \text{z}_2 > < \text{z}_3 > ...] \qquad (3)$$

Here we assume that the executable 'ee' is located in a directory in your '$PATH'. Notice that you can pass more than one redshift: The executable 'ee' accepts an arbitrary number of arguments (at least seven = six comsological parameters + one redshift) where all input parameters at position 7 and higher are considered as redshifts. The first column of the produced output corresponds to the k-mode values at which the boost factor(s) (given in the second and potentially following columns) is measured. To store this reult in a file, just use output redirection, i.e. append " > BoostFile.dat" to the command above.

Example Here we give an example for the usage of the CLI using the Euclid reference cosmology at redshifts $z_1 = 0.5$ and $z_2 = 1.2$:

```
ee 0.0219961 0.1431991 0.96 0.67 -1.0 0.83 0.5 1.2 > myResult.dat
```

The result is stored in a file called 'myResult.dat'.

Post processing WE URGE EVERYONE TO READ AT LEAST THE SECTION ABOUT PITFALLS (see below).

You may want to compute full non-linear power spectra from the boost factors you calculated with EuclidEmulator. In this case proceed as follows:

1. Produce a linear power spectrum with a Boltzmann code.
2. Interpolate the linear power spectrum (we suggest a cubic spline interpolation in log space.
3. Evaluate the interpolated liner power spectrum at the k modes of the emulated boost.
4. Multiply the boost with the resulting linear power spectrum.

The script 'example.sh' in the 'example' directory includes commands to perform these four steps (assuming you have CLASS installed already and your CLASS file explanatory.ini has never been changed). Please change the path to the directory where your CLASS executable is located. Executing 'example.sh' will sequentially compute the boost spectrum using EuclidEmulator, compute the corresponding linear power spectrum with CLASS, and finally call 'GetPnonlin.py' to compute the non-linear power spectrum.

Pitfalls of the EuclidEmulatorCLI When you want to emulate a full non-linear power spectrum, you really have to make sure that you specify the exact same cosmology for EuclidEmulatorto produce the boost factor and for the Boltzmann code you use to predict the linear power spectrum. Often, the parametrization of cosmologies used in the Boltzmann solvers is different than the one used by EuclidEmulator. Make sure the different parametrizations define the exact same cosmology!

Known differences are:

1. CAMB and CLASS use om_cdm (the cold dark matter density $\omega_{\rm cdm}$) instead of om_m (the total matter density $\omega_{\rm m}$). Make sure that the following relation is satisfied:

$$\omega_{\rm b} + \omega_{\rm cdm} = \omega_{\rm m} \tag{4}$$

   If you want to compute the non-linear power spectrum using a EuclidEmulated-boost for a real data set (like e.g. Planck2015), you may find that the reported values doe not obey the above relation. This is most likely due to the fact that there is a small contribution due to neutrinos which is taken into account in that data set but not so in this version of our emulator. Here's what you need to do:

   i. To produce the boost factor with EuclidEmulatoruse $\omega_{\rm b}$ and $\omega_{\rm m}$ as reported in the data set.
   ii. To produce the corresponding linear power spectrum use $\omega_{\rm b}$ and set the $\omega_{\rm cdm}$ parameter equal to $\omega_{\rm m} - \omega_{\rm b}$. Doing so you add the neutrino component to the CDM contribution which is the best that can be done with the current version of EuclidEmulator. Stay tuned as version 2 will allow for neutrinos to be taken into account.

2. CAMB and CLASS do usually not accept sigma_8 as a parameter for normalization of the power spectrum but rather use A_s. In order to convert these two parameters into each other in the context of using EuclidEmulator, you have to use the same conversion as is used in the EuclidEmulatorcode. Convert the parameters using the following proportionality:

$$\frac{A_s}{2.215 \cdot 10^{-9}} = \left(\frac{\sigma_8}{0.8492}\right)^2 \tag{5}$$

3 Code structure

On this page we describe the structure and the components of the EuclidEmulatorcode in great detail.

3.1 Overview

The full EuclidEmulatorcode (version 1.1) consists of 3 main parts plus some additional material all of which is structured in separate subdirectories for the sake of cleanliness. The three main parts are:

* 'libeuc', containing the code for a static C library that provides the actual emulator * 'CLI', containing the C code for a command line interface to use the emulator * 'wrapper', containing the python wrapper *e2py* of EuclidEmulator

Additional material is given in

* the 'ClassPatch' directory that contains a file 'ClassPatch.README.txt' giving information about how to install ClassPatch that is needed in order to compute fully non-linear power spectra with the python wrapper *e2py*. * the 'example' directory containing example scripts for both the CLI and the python wrapper together with plots that show some results produced with EuclidEmulator.

Below we guide the reader through all the components and explain how they work.

3.2 Background information

This section gives some additional information that may fill some logical gaps in the descriptions below. However, it is not too important at this stage. If you are going through this wiki for the first time, you can safely skip this section and come back later.

For a fully detailed discussion of how the emulator was constructed, please read the publication Knabenhans et al. (2018). Here, we only give the minimum amount of information needed to follow the explanations below.

3.2.1 EuclidEmulator is based on pre-evaluated cosmological simulations

For the construction of EuclidEmulatorversion 1.x cosmological dark matter-only simulations were run for a set of 100 cosmologies inside the 6D parameter space discussed in [Usage]https://github.com/miknab/EuclidEmulator/wiki/III)-Usage#parameter-ranges (although the exact parameter ranges used are slightly larger in order to avoid edge effects). How these 100 cosmologies were chosen is also described in the publication. From these simulations we computed the corresponding boost factors. For each simulation, the boost factor was measured at 100 different redshift values between $z_{\text{initial}} = 200.0$ and $z_{\text{final}} = 0.0$ at roughly 2000 different wave numbers (k points). The resulting data hence consisted of roughly 20 million data points in a (6+2)-dimensional space, the six cosmological parameter dimensions plus space (wave number k) and time (redshift z). Notice that the vector of k values is identical for all boost factors due to how the simulations were set up.

3.2.2 Data post-processing

1. Principal component analysis A principal component analysis was performed in order to expand the boost factor functions in a series in which each summand is a product

of a cosmology dependent coefficient ("weight") and a space-time dependent principal component (i.e. a function of k and z).

2. Polynomial chaos expansion (PCE) of principal component weights Strictly speaking, EuclidEmulatoremulates only the principal component weights. Applying sparse polynomial chaos expansion (SPCE, details can be as well found in Knabenhans et al., 2018 and references therein) EuclidEmulatorestimates the values of the principal component weights for cosmologies *in between* the 100 sampled sets of cosmological parameters for which actual simulations were run. A SPCE is a kind of series expansion in terms of ortho-normal basis functions (in our case these are normalized Legendre polynomials).

To summarize, the emulation itself is the very (double) series expansion described above. To evaluate it, one hence needs the principal component functions, the set of multi-indices specifying which ortho-normal basis functions have to be used and their coefficients. All this data is compiled in the binary data file 'ee.dat' (in the 'libeuc' subdirectory).

3.3 libeuc: the actual emulator

In the 'libeuc' subdirectory, there are two C-code (.c) files, each with a corresponding header (.h) file:

- 'cosmo.c'
- 'EuclidEmulator.c'
- 'cosmo.h'
- 'EuclidEmulator.h.in' (this file is a configurable header, hence the extension ".h.in").

and in addition a binary data file called 'ee.dat'. Compiling these codes results in the static library 'libeuc.a' which is the core of EuclidEmulator. This library is used by both the CLI and the *e2py* wrapper.

'EuclidEmulator.c' The code in this file consists of a single function (which you will NEVER have to directly interact with)

```
void EucEmu(double *CosmoParams, double *Redshifts, int len_z, double **kVa
```

that emulates boost factors, non-linear corrections to matter power spectra. The arguments of this function are:

1. input parameters:
   - '*CosmoParams': a pointer to the input set of cosmological parameters
   - '*Redshifts': a pointer to the input set of redshift values
   - 'len_z': the number of input redshift values
2. containers for output parameters:

- ‘**kVals‘: container for the output vector containing the wave numbers
- ‘*nkVals‘: container for the length of the output vector containing the wave numbers
- ‘**Boost‘: container for the output boost factor values
- ‘*nBoost‘: container for the length of the output boost factor values

In this function, first of all, the data file ‘ee.dat‘ is mapped into your computer's memory (as the data file is relatively heavy, this speeds up the reading process). Once this is done, the Legendre polynomials are computed up the maximally required polynomial order and evaluated for specified input cosmology. This step requires GSL.

Next, the actual emulation is performed by plugging in all the data into the SPCE in order to compute the principal component weights. We then multiply them with the principal component functions.

There is an additional interpolation step at the end of this function that allows for any input redshift value (as long as it respects the range). This is necessary as we only have simulation data from a discrete set of 100 different redshifts. To this end, we need information about the time evolution in the given input cosmology which we get from functions in ‘cosmo.c‘.

‘cosmo.c‘ This file is part of the [pkdgrav3]www.pkdgrav.com code (D. Potter & J. Stadel) that was used to produce the cosmological N-body simulations underlying this emulator. It contains (amongst others) the ‘csmExp2Time‘ function that is relevant for EuclidEmulator: It converts for a given cosmology the cosmological scale factor a into time.

3.4 CLI: the command line interface

This subdirectory contains the code for the command line interface to the core library ‘libeuc.a‘. This interface consists of two code files:

* ‘EuclidEmulator.h‘: The header file for ‘EuclidEmulator.c‘ * ‘master.c‘: contains the ‘main‘ function callable from the command line

When compiled, the resulting executable is named ‘ee‘. If you install the CLI, ‘ee‘ is copied to your installation directory. The ‘EuclidEmulator.h‘ file is the same header file as described above in the section about the libeuc library. So we will focus here on the file ‘master.c‘. The main function in this file reads in the command line parameters. It expects seven or more parameters corresponding to six cosmological parameters plus an arbitrary number of redshift values. Hence the call

```
ee 0.0219961 0.1431991 0.96 0.67 -1.0 0.83 0.0
```

would compute the boost factor at redshift z=0.0 only while

```
ee 0.0219961 0.1431991 0.96 0.67 -1.0 0.83 0.0 1.0 2.0 3.0
```

would compute four boost factors at redshifts z=0.0, 1.0, 2.0 and 3.0. By default, ‘ee‘ prints the results to standard output and in this example looks as follows (we just print to first few lines):

```
# \textsc{EuclidEmulator} Version 1.1 Copyright (C) 2018 Mischa Knabenhans &
# This program comes with ABSOLUTELY NO WARRANTY; for details type 'show w'
# This is free software, and you are welcome to redistribute it
# under certain conditions; type 'show c' for details.
#
# Units:
#     k has units h/Mpc
#     B is dimensionless
#
# Cosmology:
#    dOmega0: 0.31900000
#    dOmegaRad: 0.00009320
#    dOmegaDE: 0.68090680
#
# Computed 1 of 4 boost factors.
# Computed 2 of 4 boost factors.
# Computed 3 of 4 boost factors.
# Computed 4 of 4 boost factors.
#
# ================================
# k             B(z=0.000)      B(z=1.000)      B(z=2.000)      B(z=3.000)
#
0.006872        0.999476        0.999866        0.999962        0.999992
0.011990        0.997057        0.998941        0.999538        0.999784
0.017098        0.994555        0.998038        0.999095        0.999490
0.022180        0.991291        0.996413        0.998041        0.998703
0.027419        0.989696        0.995741        0.997654        0.998447
0.032346        0.987888        0.995588        0.998008        0.998914
0.037381        0.986229        0.994453        0.997135        0.998256
0.042447        0.984578        0.994398        0.997508        0.998717
```

If you want to store the results in a file, please use output redirection to do so, i.e. for instance

```
ee 0.0219961 0.1431991 0.96 0.67 -1.0 0.83 0.0 1.0 2.0 3.0 > boost_z0.0.dat
```

Notice that everything above and including the line '# ================================' is printed to stderr such that only the resulting data is redirected to the file 'boost_z0.0.dat'.

3.5 wrapper: the python wrapper

Next, the components of the python wrapper *e2py* shall be described in great detail. It shall be emphasized once more that we highly encourage all users to use this wrapper in order to take advantage of many utilities that the CLI does not provide.

3.5.1 The Cython layer in 'EuclidEmulator_BackEnd.pyx'

First it shall be made clear that also the python wrapper *e2py* uses the C library libeuc in order to compute the boost factors. The interface between python and C is written in

the hybrid language Cython and is found in the file 'EuclidEmulator_BackEnd.pyx'. The main purpose of this Cython code is to transform back and forth between python an C data structures and types.

The most important function in 'EuclidEmulator_BackEnd.pyx' is 'emu_boost' with the signature

```
def emu\_boost(cosmo_par_array, redshift_array)
```

This means, it takes a list or a numpy array with the cosmological parameters as a first and a list or a numpy array with a number of redshift values as a second argument. The function converts these iterables into cython memory views. Moreover, in this function a the data containers for the output data described in the section about [EuclidEmulator.c]https://github.com/miknab/EuclidEmulator/wiki/IV)-Code-Structure#euclidemulatorc are defined.

Remark: While the chosen implementation with pointers to pointers for output variables might look a bit cumbersome at first sight, we chose this approach because it allows for ignorance of the size of the output data of the function 'EucEmu' in 'EuclidEmulator.c'. The length of the k vector and the boost vector is hard coded in the binary file 'ee.dat' which is only read in inside 'EucEmu'. The chosen strategy hence allows to update 'ee.dat' without having to rewrite the code.

In a next step. the cython program calls the 'EucEmu' function described above. The output data containers with the k vectors and the boost values are converted again into cython memory views and put into a class 'Py_ee_class' which takes care of both, conversion to python-understandable data types and proper freeing of memory.

Compilation of 'EuclidEmulator_BackEnd.pyx' results in a shared object file called 'EuclidEmulator_BackEnd.so' is moved to the libraries related to your python interpreter. This shared object is the actual callable wrapper that is used at the heart of the python package *e2py* described below.

3.5.2 The e2py python package

With the *e2py* package we pursue two goals:

- simplifying work flows related to EuclidEmulator
- adding utilities to the core functionality of EuclidEmulator

The python scripts of *e2py* can be categorized into two groups: the scripts containing functions the users should interact with (let's call them *public*) and scripts that contain functions that shall only be used by other functions and hence are not written for direct access by users (let's call them *private*).

Notice: We follow the common python standard by giving all "private" python scripts a name starting with a underscore (_).

Most private scripts are "hidden" inside the module '_internal' with the single exception of '_ee_lens.py' which does contain functionalities that shall be used by users in the future but that are not ready yet.

In this subsection, we will go explain the two public python scripts in depth both of which are imported upon execution of

```
import e2py
```

As a result, all functions defined in either one of these two python scripts are available as functions of *e2py*, for instance it is possible to compute the boost via ‘e2py.get_boost‘ and it is not necessary to call ‘e2py.ee_observables.get_boost‘.

‘ee_input.py‘ This module contains the following set of functions:

- ‘check_param_range‘: checks if a set of cosmological parameters obey the parameter ranges expected by EuclidEmulator.
- ‘read_parfile‘: reads in a parameter file and constructs python dictionaries from its content
- ‘emu_to_class‘: converts EuclidEmulator-formatted parameters into Class-formatted parameters
- ‘class_to_emu‘: converts Class-formatted parameters into EuclidEmulator-formatted parameters

—— | **‘check_param_range‘** | | | ————- | ———————————- | | Signature | check_param_range(par_dict, csm_index=0) | | Description | This function checks if all parameters in the dictionary ’par_dict’ passed to this function obey the limits set by the emulator.| | Input | par_dict is a dictionary containing cosmological parameters as expected by EuclidEmulator| | Input types | type(par_dict)=dict, type(csm_index)=int | | Output | None | | Output types | None | | Effect | If all parameter ranges are obeyed, this function has no effect. Otherwise, a ‘ValueError‘ is risen. |

Example:

```
In [1]: import e2py
In [2]: MyCosmo = {'om_b': 0.0219961, 'om_m': 0.1431991, 'n_s': 0.96, 'h':
In [3]: e2py.check\_param\_range(MyCosmo)
```

This does not produce any output, as all parameters are inside the bounds. On the other hand

```
In [4]: MyCosmo2 = {'om_b': 0.0, 'om_m': 0.1431991, 'n_s': 0.96, 'h': 0.67,
In [5]: e2py.check\_param\_range(MyCosmo2)
```

leads to the output

```
Out[5]: ValueError: Parameter range violation in cosmology 0:
        om_b is set to 0.000000, but should be in the interval
        [0.0217, 0.0233].
```

—-

| **‘read_parfile‘** | | | ————- | ———————————- | | Signature | read_parfile(filename, sep=’,’) | | Description | This function reads in parameter files containing cosmological parameters. By default, this function assumes that the cosmological parameters in the parameter file are comma-separated. This behaviour can be modified by explicitly giving a separator. | | Input | ‘filename‘ is a parameter file, ‘sep‘ is the separator between the ”cells” in the parameter file | | Input types | type(filename)=str,
 type(sep)=str | | Output | A python dictionary or a list of python dictionaries with the keys being given by the first line in the parameter file and values corresponding to the

numbers in the parameter file. | | Output types | type(output)=dict (if parameter file contains only one line of parameter values),
 type(output)=list (if parameter file contains several lines of parameter values) |

Example:

Assume there is a parameter file called 'TestParFile.csv' with the following content

om_b	om_m	n_s	h	w_0	sigma_8
0.0219961	0.1431991	0.96	0.67	−1.00	0.83
0.02312	0.1501	0.956	0.72	−0.89	0.81
0.02173	0.1399	0.968	0.71	−0.95	0.80

Notice that the separator is given by a varying number of whitespaces. So, we read in this file as follows

```
In [1]: import e2py
In [2]: e2py.read\_parfile('TestParFile.csv', sep='\s+')
```

where as a separator we pass a regular expression matching any number of white spaces (at least one). The result is given by:

```
Out[2]: [{'h': 0.67000000000000004,
          'n_s': 0.95999999999999996,
          'om_b': 0.021996099999999998,
          'om_m': 0.1431991,
          'sigma_8': 0.82999999999999996,
          'w_0': -1.0},
         {'h': 0.71999999999999997,
          'n_s': 0.95599999999999996,
          'om_b': 0.023119999999999998,
          'om_m': 0.15010000000000001,
          'sigma_8': 0.81000000000000005,
          'w_0': -0.89000000000000001},
         {'h': 0.70999999999999996,
          'n_s': 0.96799999999999997,
          'om_b': 0.021730000000000003,
          'om_m': 0.1399,
          'sigma_8': 0.80000000000000004,
          'w_0': -0.94999999999999996}]
```

—-

| **'emu_to_class'** | | | ———————— | ——- | | Signature | emu_to_class(emu_pars_dict) | | Description | This function converts the set of parameters accepted by EuclidEmulatorinto a set of parameters accepted by CLASS and CAMB | | Input | 'emu_pars_dict' is a python dictionary with six cosmological parameters. The expected keys are:
 'om_b' (lowercase baryon density parameter),
 'om_m' (lowercase total matter density parameter),
 'n_s' (spectral index),
 'h' (dimensionless Hubble parameter),
 'w_0' (dark energy equation of state parameter),
 'sigma_8' (overdensity fluctuation variance). | | Input type | type(emu_pars_dict)=dict | | Output | a python dictionary with cosmological parameters expected by class. The keys are:
 'omega_b' (lowercase baryon density parameter),
 'omega_cdm' (lowercase cold

dark matter density parameter),
 ‘n_s‘ (spectral index),
 ‘h‘ (dimensionless Hubble parameter),
 ‘w0_fld‘ (dark energy equation of state parameter),
 ‘A_s‘ (spectral amplitude). | | Output type | type(output)=dict |

Example:

Define a cosmology, pass it to ‘emu_to_class‘ and obtain a class-formatted dictionary with cosmological parameters:

```
In [1]: import e2py
In [2]: MyCosmo = {'om_b': 0.0219961, 'om_m': 0.1431991, 'n_s': 0.96, 'h':
In [3]: e2py.emu\_to\_class(MyCosmo)
```

This is the resulting output:

```
Out[3]: {'A_s': 2.1139801637017173e-09,
         'h': 0.67,
         'n_s': 0.96,
         'omega_b': 0.0219961,
         'omega_cdm': 0.12120299999999999,
         'w0_fld': -1.0}
```

—-

| **‘class_to_emu‘** | | | ——————— |——- | | Signature | class_to_emu(class_pars_dict) | | Description | This function converts the set of parameters accepted by CLASS and CAMB into a set of parameters accepted by EuclidEmulator| | Input | ‘class_pars_dict‘ is a python dictionary with six cosmological parameters. The expected keys are:
 ‘omega_b‘ (lowercase baryon density parameter),
 ‘omega_cdm‘ (lowercase cold dark matter density parameter),
 ‘n_s‘ (spectral index),
 ‘h‘ (dimensionless Hubble parameter),
 ‘w0_fld‘ (dark energy equation of state parameter),
 ‘A_s‘ (spectral amplitude).| | Input type | type(class_pars_dict)=dict | | Output | a python dictionary with key corresponding to cosmological parameters expected by class. The keys are:
 ‘om_b‘ (lowercase baryon density parameter),
 ‘om_m‘ (lowercase total matter density parameter),
 ‘n_s‘ (spectral index),
 ‘h‘ (dimensionless Hubble parameter),
 ‘w_0‘ (dark energy equation of state parameter),
 ‘sigma_8‘ (overdensity fluctuation variance). | | Output type | type(output)=dict |

Example:

Define a cosmology in CLASS-format, pass it to the function ‘class_to_emu‘ and obtain a EuclidEmulator-formatted python dictionary with cosmological parameters:

```
In [1]: import e2py
In [2]: MyCosmoClass={'A_s': 2.1e-09, 'h': 0.72, 'n_s': 0.96, 'omega_b': 0.
In [3]: e2py.class\_to\_emu(MyCosmoClass)
```

The output the looks like:

```
Out[3]: {'h': 0.72,
         'n_s': 0.96,
         'om_b': 0.02222,
         'om_m': 0.14342,
         'sigma_8': 0.827250971441976,
         'w_0': -1.0}
```

‘ee_observables.py‘ This module contains the following set of functions:

- `get_boost`: calls the emulator core (via the cython layer) to compute the boost factor
- `get_plin`: calls Class to compute the linear power spectrum
- `get_pnonlin`: combines linear power spectra and boost factor into a non-linear power spectrum
- ‘sgaldist‘: computes a source galaxy distribution (used for weak lensing studies)

—- | ** `get_boost`** | | | ———————— |——- | | Signature | get_boost(emu_pars_dict, redshifts) | | Description | This function computes (emulates) the boost factors for the given cosmology at all passed redshift values. This function runs independently of CLASS. | | Input | ‘emu_pars_dict‘ contains the cosmology and ‘redshifts‘ the redshifts at which the boost shall be computed. They keys expected in ‘emu_pars_dict‘ are:
 ‘om_b‘ (lowercase baryon density parameter),
 ‘om_m‘ (lowercase total matter density parameter),
 ‘n_s‘ (spectral index),
 ‘h‘ (dimensionless Hubble parameter),
 ‘w_0‘ (dark energy equation of state parameter),
 ‘sigma_8‘ (overdensity fluctuation variance)| | Input type | type(emu_pars_dict)=dict
 type(redshifts)=int, float, list or numpy.ndarray | | Output | a python dictionary with keys ’B’ and ’k’ corresponding to the boost factor values and the k values at which the boost is computed. | | Output type | type(output)=dict |

Example 1:

Let’s first have a look how the boost factor for a given cosmology is computed at one single redshift:

```
In [1]: import e2py
In [2]: MyCosmo = {'om_b': 0.0219961, 'om_m': 0.1431991, 'n_s': 0.96, 'h':
In [3]: e2py.get\_boost(MyCosmo,0.5)
```

The output is:

```
Out[3]: # Computed 1 of 1 boost factors.
        {'B': array([ 0.9997385 ,  0.99826641,  0.99680089, ..., 22.8561094
                22.85609302, 22.85952702]),
         'k': array([0.00687215, 0.01199   , 0.0170982 , ..., 5.51663
, 5.52166    ,
                5.52669    ])}
```

Example 2:

Now we’ll show how the boost factor for a given cosmology is computed at several different redshifts with only one call of `get_boost`:

```
In [1]: import e2py
In [2]: import numpy as np

In [3]: MyCosmo = {'om_b': 0.0219961, 'om_m': 0.1431991, 'n_s': 0.96, 'h':
In [4]: zvec = np.linspace(0.0,5.0,5)

In [5]: e2py.get\_boost(MyCosmo,zvec)
```

The output is:

```
Out[5]: # Computed 1 of 5 boost factors.
        # Computed 2 of 5 boost factors.
        # Computed 3 of 5 boost factors.
        # Computed 4 of 5 boost factors.
        # Computed 5 of 5 boost factors.
        {'B': {'z0': array([ 0.99947578,  0.99705664,  0.99455529, ..., 29.
,
                29.98188102, 29.98566645]),
          'z1': array([ 0.9999065 ,  0.99915842,  0.99842694, ..., 18.17388
                18.17112057, 18.17527751]),
          'z2': array([ 0.99998278,  0.99968334,  0.99933467, ..., 10.50938
                10.5208513 , 10.52581194]),
          'z3': array([0.9999999 , 0.99988819, 0.99963588, ..., 5.27193775,
                 5.28529626]),
          'z4': array([0.99999642, 0.99999246, 0.99975585, ..., 2.88413457,
                 2.8879599 ])},
         'k': array([0.00687215, 0.01199   , 0.0170982 , ..., 5.51663
, 5.52166    ,
                5.52669    ])}
```

The result is now a nested python dictionary. While the value for the key 'k' is always a single np.ndarray, the value for the key 'B' is now a dictionary itself with keys 'z0','z1','z2', etc. where 'z0' refers to the first, 'z1' to the second redshift etc. in the array 'zvec'.

—- | ** `get_plin`** | | | ———— |—- | | Signature | get_plin(emu_pars_dict, kvec, redshifts) | | Description | This function passes the input parameters on to classee (or if not available to classy) and tells it to compute the linear power spectrum for the given cosmology, at the given k modes and redshifts.
 IMPORTANT: This function relies on an installation of classee (or at least classy). If none of these CLASS python wrappers is installed, this function does not work. If you only have the official classy installed, there are many cases were this function will fail (e.g. if 'kvec' contains more than 31 elements). Classee is the wrapper of a patched version of CLASS that allows for instance also for longer 'kvec'.| | Input | 'emu_pars_dict' contains the cosmology and 'redshifts' the redshifts at which the boost shall be computed. They keys expected in 'emu_pars_dict' are:
 'om_b' (lowercase baryon density parameter),
 'om_m' (lowercase total matter density parameter),
 'n_s' (spectral index),
 'h' (dimensionless Hubble parameter),
 'w_0' (dark energy equation of state parameter),
 'sigma_8' (overdensity fluctuation variance)

 'kvec' contains the k values and 'redshifts' the redshifts at which the linear power spectrum shall be computed. | | Input type | type(emu_pars_dict)=dict
 type(kvec)=int, float, list or numpy.ndarray
 type(redshifts)=int, float, list or numpy.ndarray | | Output | a python dictionary with keys 'k' and 'P_lin' corresponding to the k values at which the linear power spectrum (stored in 'P_lin') is computed. | | Output type | type(output)=dict |

Example 1:

Let's first have a look how the linear power spectrum for a given cosmology is computed at one single redshift:

```
In [1]: import e2py
In [2]: import numpy as np

In [3]: MyCosmo = {'om_b': 0.0219961, 'om_m': 0.1431991, 'n_s': 0.96, 'h':
In [4]: kvec = np.logspace(-2,1,2000)
In [5]: e2py.get\_plin(MyCosmo,kvec,0.5)
```

The output looks like:

```
Out[5]: {'P_lin': array([1.30502632e+04, 1.30684970e+04, 1.30866665e+04, ..
                 1.39711652e-01, 1.38453422e-01, 1.37206384e-01]),
          'k': array([ 0.01      ,  0.01003462,  0.01006935, ...,
9.93112617,
                 9.96550358, 10.        ])}
```

Example 2:

Again, we'll show how the output changes if multiple redshift values are passed to the function `get_plin`:

```
In [1]: import e2py
In [2]: import numpy as np

In [3]: MyCosmo = {'om_b': 0.0219961, 'om_m': 0.1431991, 'n_s': 0.96, 'h':
In [4]: kvec = np.logspace(-2,1,2000)
In [5]: zvec = np.linspace(0.0,5.0,5)

In [6]: e2py.get\_plin(MyCosmo,zvec)
```

The output is:

```
Out[6]: {'P_lin': {'z0': array([2.21346334e+04, 2.21655713e+04, 2.21963998e
                 2.36966378e-01, 2.34832272e-01, 2.32717149e-01]),
          'z1': array([6.57324292e+03, 6.58242638e+03, 6.59157715e+03, ...,
                 7.03717800e-02, 6.97380242e-02, 6.91099052e-02]),
          'z2': array([2.84560139e+03, 2.84957535e+03, 2.85353506e+03, ...,
                 3.04657578e-02, 3.01913970e-02, 2.99194763e-02]),
          'z3': array([1.56214899e+03, 1.56433189e+03, 1.56650696e+03, ...,
                 1.67256033e-02, 1.65749837e-02, 1.64257038e-02]),
          'z4': array([9.83045149e+02, 9.84418708e+02, 9.85787346e+02, ...,
                 1.05259034e-02, 1.04311160e-02, 1.03371715e-02])},
          'k': array([ 0.01      ,  0.01003462,  0.01006935, ...,
9.93112617,
                 9.96550358, 10.        ])}
```

The result is now again a nested python dictionary. While the value for the key 'k' is always a single np.ndarray, the value for the key 'P_lin' is now a dictionary itself with keys 'z0','z1','z2', etc. where 'z0' refers to the first, 'z1' to the second redshift etc. in the array 'zvec'.

—- | ** `get_pnonlin`** | | | ————— |——- | | Signature | get_pnonlin(emu_pars_dict, redshifts) | | Description | This function passes the input parameters

on to `get_boost` from where a boost factor and the corresponding vector of k values is obtained. The cosmological parameters, the redshifts and now also the k vector are passed to `get_plin` where a linear power spectrum is computed. The resulting linear power spectrum is multiplied with the boost factor resulting in a fully non-linear power spectrum
 IMPORTANT: This function relies on an installation of classee (or at least classy) because it calls `get_plin`. If none of these CLASS python wrappers is installed, this function does not work. If you only have the official classy installed, there are many cases were this function will fail (e.g. if 'kvec' contains more than 31 elements). Classee is the wrapper of a patched version of CLASS that allows for instance also for longer 'kvec'.| | Input | 'emu_pars_dict' contains the cosmology and 'redshifts' the redshifts at which the boost shall be computed. They keys expected in 'emu_pars_dict' are:
 'om_b' (lowercase baryon density parameter),
 'om_m' (lowercase total matter density parameter),
 'n_s' (spectral index),
 'h' (dimensionless Hubble parameter),
 'w_0' (dark energy equation of state parameter),
 'sigma_8' (overdensity fluctuation variance)

 'redshifts' contains the redshifts at which the linear power spectrum shall be computed. | | Input type | type(emu_pars_dict)=dict
 type(redshifts)=int, float, list or numpy.ndarray | | Output | a python dictionary with keys 'k', 'B', 'P_lin' and 'P_nonlin' is returned. | | Output type | type(output)=dict |

Example 1:

Let's first have a look once more how the linear power spectrum for a given cosmology is computed at one single redshift:

```
In [1]: import e2py

In [2]: MyCosmo = {'om_b': 0.0219961, 'om_m': 0.1431991, 'n_s': 0.96, 'h':
In [3]: e2py.get\_pnonlin(MyCosmo,kvec,0.5)
```

The output looks like:

```
Out[3]: {'B': array([ 0.9997385 ,  0.99826641,  0.99680089, ..., 22.8561094
                22.85609302, 22.85952702]),
         'P_lin': array([1.08142879e+04, 1.39034360e+04, 1.46366161e+04, ..
                6.40780294e-01, 6.39284317e-01, 6.37793169e-01]),
         'P_nonlin': array([1.08114600e+04, 1.38793332e+04, 1.45897920e+04,
                1.46457445e+01, 1.46115418e+01, 1.45796502e+01]),
         'k': array([0.00687215, 0.01199   , 0.0170982 , ..., 5.51663
, 5.52166    ,
                5.52669    ])}
```

Example 2:

Again, we'll show how the output changes if multiple redshift values are passed to the function `get_plin`:

```
In [1]: import e2py
In [2]: import numpy as np

In [3]: MyCosmo = {'om_b': 0.0219961, 'om_m': 0.1431991, 'n_s': 0.96, 'h':
In [4]: zvec = np.linspace(0.0,5.0,5)

In [5]: e2py.get\_pnonlin(MyCosmo,zvec)
```

The output is:

```
Out[5]: {'B': {'z0': array([ 0.99947578,  0.99705664,  0.99455529, ..., 29.
,
                29.98188102, 29.98566645]),
         'z1': array([ 0.9999065 ,  0.99915842,  0.99842694, ..., 18.17388
                18.17112057, 18.17527751]),
         'z2': array([ 0.99998278,  0.99968334,  0.99933467, ..., 10.50938
                10.5208513 , 10.52581194]),
         'z3': array([0.9999999 , 0.99988819, 0.99963588, ..., 5.27193775,
                5.28529626]),
         'z4': array([0.99999642, 0.99999246, 0.99975585, ..., 2.88413457,
                2.8879599 ])},
        'P_lin': {'z0': array([1.83423598e+04, 2.35822883e+04, 2.48255618e
                1.08690082e+00, 1.08436409e+00, 1.08183552e+00]),
         'z1': array([5.44710892e+03, 7.00291590e+03, 7.37211709e+03, ...,
                3.22771277e-01, 3.22017861e-01, 3.21266874e-01]),
         'z2': array([2.35798326e+03, 3.03156862e+03, 3.19139872e+03, ...,
                1.39740090e-01, 1.39414032e-01, 1.39089020e-01]),
         'z3': array([1.29443630e+03, 1.66424948e+03, 1.75200280e+03, ...,
                7.67190550e-02, 7.65401000e-02, 7.63617184e-02]),
         'z4': array([8.14578373e+02, 1.04731950e+03, 1.10255481e+03, ...,
                4.82807564e-02, 4.81681378e-02, 4.80558800e-02])},
        'P_nonlin': {'z0': array([18332.74443775, 23512.87712884, 24690.39
               32.58556583,    32.51127505,    32.43955915]),
         'z1': array([5.44659961e+03, 6.99702237e+03, 7.36052028e+03, ...,
                5.86600669e+00, 5.85142539e+00, 5.83911459e+00]),
         'z2': array([2.35794265e+03, 3.03060864e+03, 3.18927539e+03, ...,
                1.46858243e+00, 1.46675429e+00, 1.46402487e+00]),
         'z3': array([1.29443617e+03, 1.66406340e+03, 1.75136487e+03, ...,
                4.04458082e-01, 4.04041617e-01, 4.03594304e-01]),
         'z4': array([8.14575458e+02, 1.04731161e+03, 1.10228563e+03, ...,
                1.39248198e-01, 1.39003107e-01, 1.38783454e-01])},
         'k': array([0.00687215, 0.01199   , 0.0170982 , ..., 5.51663
, 5.52166    ,
                5.52669    ])}
```

The result is now again a nested python dictionary. While the value for the key ‘k‘ is always a single np.ndarray, the value for the keys ‘B‘, ‘P_lin‘ and ‘P_nonlin‘ are now a dictionary themselves with keys ‘z0‘,‘z1‘,‘z2‘, etc. where ‘z0‘ refers to the first, ‘z1‘ to the second redshift etc. in the array ‘zvec‘.

—

| **‘sgaldist‘** | | | ————- | ——————————- | | Signature | sgaldist(alpha=2.0, beta=1.5, z_mean=0.9) | | Description | This function takes three (optional) keyword arguments (or any subset thereof) and returns a source galaxy distribution according to the standard distribution function given by

 $n(z; alpha, beta, z\_mean) = \left(\frac{z}{z_0}\right)^{\text{alpha}} \exp[-\left(\frac{z}{z_0}\right)^{beta}]$

where z_0 = z_mean/1.412. Notice that the default

values of alpha, beta and z_mean are as listed in 'Signature'.| | Input | the input parameters are the parameters of the distribution function $n(z)$ mentioned above.| | Input types | type(alpha)=float,
type(beta)=float,
type(z_mean)=float| | Output | This function returns an object that is similar to a function object in the sense that it takes arguments. To be precise, the returned object takes a numpy.ndarray of redshifts as arguments and computes the corresponding source galaxy distribution in redshift based on the passed parameters. | | Output types | type(output)=e2py._internal._ee_aux.Function (EuclidEmulator internal type) |

Example:

This example shows how one obtains a full source galaxy distribution data set for three non-default parameters:

```
In [1]: import e2py
In [2]: import numpy as np

In [3]: n_func = e2py.sgaldist(alpha=2.1, beta=1.45, z\_mean=0.95)
```

Printing 'n_func' just tells us that it is an object of a specific type stored at a certain address:

```
In [4]: n_func
Out[4]: <e2py._internal.\_ee\_aux.Function at 0x7f18a3f3fb10>
```

In order to get actual numbers (that we could plot, for instance), we have to evaluate that returned function at a set of redshift values:

```
In [5]: zvec = np.linspace(0.0,5.0,2000)
In [6]: nvec = n_func(zvec)
```

The return value 'nvec' is now a numpy.array:

```
In [7]: nvec
Out[7]: array([0.00000000e+00, 1.59586335e-05, 6.83806371e-05, ...,
               1.53668354e-06, 1.51799344e-06, 1.49952539e-06])
```

3.6 ClassPatch

Inside this subdirectory you find a file 'ClassPatch.README.txt' that gives some hints on how to install *classee*, the python wrapper of that version of the CLASS code that is patches such that it can be used by EuclidEmulator. This directory may be used as a download directory for the ClassPatch code (see: https://github.com/miknab/ClassPatch). The thought behind this is that this patch is tailored to be used by EuclidEmulatorso it makes sense to keep these two codes at the same place. But of course, this is not mandatory at all and you can download, build and install ClassPatch wherever you want on your system. As long as you follow the installation instruction of the ClassPatch code, it should always work (please report if it does not).

3.7 examples

In the subdirectory 'examples' one can find the following files:

- ‘AVG_EuclidReference.wRad.00100.pk‘: This file contains the fully non-linear power spectrum of the Euclid Reference cosmology at redshift z=0 produced in a N-body simulation run with pkdgrav3.
- ‘ExamplePlot.pdf‘: This plot corresponds to the blue curve in Figure 9 in Knabenhans et al., 2018. It can be reproduced by the script ‘ProducePublicationPlot.py‘
- ‘example.sh‘: This is a bash script that showcases how to work with the EuclidEmulatorCLI. It computes the boost factor for a few redshifts and stores the result into a data file named ”EXAMPLE_EucRefBoost.dat”
- ‘ProducePublicationPlot.py‘: Script to produce ‘ExamplePlot.pdf‘.
- ‘PythonExamplePlot.pdf‘: This plot is the result of the python test script ‘test.py‘.
- ‘TestParFile.csv‘: This file is an example parameter file. It is written in a way that can be easily understood by both humans and the EuclidEmulatorfunction (in contrast to a pure comma-separated file that would be hard to read for humans).
- ‘test.py‘: This is a full script showing how to work with EuclidEmulator. The result of this script is a data file stored in a directory ‘DataOutput‘ (which is created automatically if it does not already exist) and the plot ‘PythonExamplePlot.pdf‘.