jax-cosmo

Feb 05, 2023

Tutorials

A differentiable and GPU accelerated cosmology library in JAX.

Checkout the jax-cosmo GitHub page.

CHAPTER 1

Install jax-cosmo

Because jax-cosmo is written in plain Python, installing it is trivial:

```
$ pip install jax-cosmo
```

That's all there is to it.

1.1 Introduction to jax-cosmo

Authors: - [@EiffL](https://github.com/EiffL) (Francois Lanusse)

1.1.1 Overview

`jax-cosmo` brings the power of automatic differentiation and XLA execution to cosmological computations, all the while preserving the readability and human friendliness of Python / NumPy.

This is made possible by the JAX framework, which can be summarised as JAX = NumPy + autograd + GPU/TPU. We encourage the interested reader to follow this introduction to JAX but it will not be necessary to follow this notebook.

1.1.2 Learning objectives

In this short introduction we will cover: - How to define computations of **2pt functions** - How to execute these computations on **GPU** (spoiler alert, you actually don't need to do anything, it happens automatically) - How to **take derivatives** of any quantities by automatic differentation - And finally, how to piece all of this together for efficient and reliable **Fisher matrices**.

Installing and importing jax-cosmo

One of the important aspects of `jax-cosmo` is that it is entirely Python-based so it can trivially be installed without compiling or downloading any third-party tools.

Here is how to install the current release on your system:

[1]:
```
# Installing jax-cosmo
!pip install --quiet jax-cosmo
```

```
     || 286kB 8.5MB/s
  Building wheel for jax-cosmo (setup.py) ... done
```

For efficient computation on GPU (if you have one), you might want to make sure that JAX itself is installed with the proper GPU-enabled backend. See here for more instructions.

Now that `jax-cosmo` is installed, let's import it along with JAX tools:

[2]:
```
%pylab inline
import jax
import jax_cosmo as jc
import jax.numpy as np

print("JAX version:", jax.__version__)
print("jax-cosmo version:", jc.__version__)
```

```
Populating the interactive namespace from numpy and matplotlib
JAX version: 0.2.0
jax-cosmo version: 0.1rc7
```

Note that we import the JAX version of NumPy here. That's all that you have to do, any numpy functions you will use afterwards will be JAX-accelerated and differentiable.

And for the purpose of this tutorial we also define a few plotting functions in the cell bellow, please run it.

[3]:
```
#@title Defining some plotting functions [run me]

import matplotlib.pyplot as plt
from matplotlib.patches import Ellipse

def plot_contours(fisher, pos,  nstd=1., ax=None, **kwargs):
  """
  Plot 2D parameter contours given a Hessian matrix of the likelihood
  """

  def eigsorted(cov):
    vals, vecs = linalg.eigh(cov)
    order = vals.argsort()[::-1]
    return vals[order], vecs[:, order]

  mat = fisher
  cov = np.linalg.inv(mat)
  sigma_marg = lambda i: np.sqrt(cov[i, i])

  if ax is None:
      ax = plt.gca()

  vals, vecs = eigsorted(cov)
  theta = degrees(np.arctan2(*vecs[:, 0][::-1]))
```

(continues on next page)

(continued from previous page)

```
  # Width and height are "full" widths, not radius
  width, height = 2 * nstd * sqrt(vals)
  ellip = Ellipse(xy=pos, width=width,
                  height=height, angle=theta, **kwargs)

  ax.add_artist(ellip)
  sz = max(width, height)
  s1 = 1.5*nstd*sigma_marg(0)
  s2 = 1.5*nstd*sigma_marg(1)
  ax.set_xlim(pos[0] - s1, pos[0] + s1)
  ax.set_ylim(pos[1] - s2, pos[1] + s2)
  plt.draw()
  return ellip
```

Defining a Cosmology and computing background quantities

We'll beginning with the basics, let's define a cosmology:

[4]:
```
# Create a cosmology with default parameters
cosmo = jc.Planck15()
```

[5]:
```
# Alternatively we can override some of the defaults
cosmo_modified = jc.Planck15(h=0.7)
```

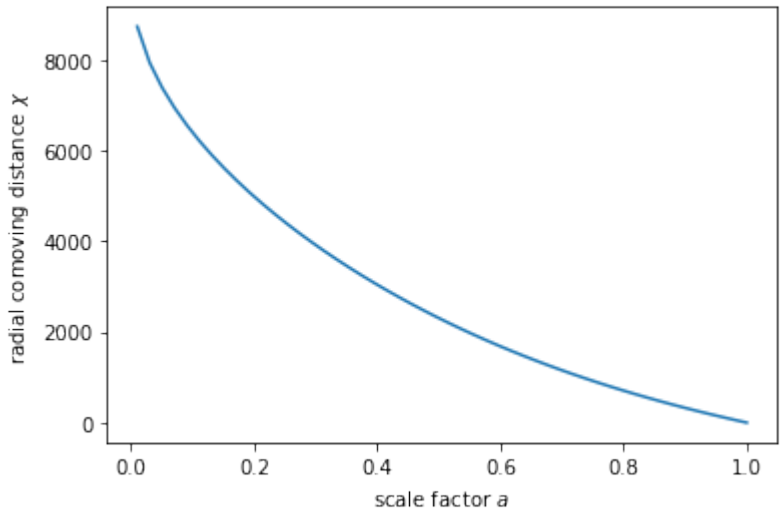
[6]:
```
# Parameters can be easily accessed from the cosmology object
cosmo.h
```

[6]:
```
0.6774
```

All background quantities can be computed from the `jax_cosmo.background` module, they typically take the cosmology as first argument, and a scale factor argument if they are not constant.

[7]:
```
# Let's define a range of scale factors
a = np.linspace(0.01, 1.)

# And compute the comoving distance for these scale factors
chi = jc.background.radial_comoving_distance(cosmo, a)

# We can now plot the results:
plot(a, chi)
xlabel(r'scale factor $a$')
ylabel(r'radial comoving distance $\chi$');
```

```
/usr/local/lib/python3.6/dist-packages/jax/lax/lax.py:6181: UserWarning: Explicitly␣
↪requested dtype <class 'jax.numpy.lax_numpy.int64'> requested in astype is not␣
↪available, and will be truncated to dtype int32. To enable more dtypes, set the jax_
↪enable_x64 configuration option or the JAX_ENABLE_X64 shell environment variable.␣
↪See https://github.com/google/jax#current-gotchas for more.
  warnings.warn(msg.format(dtype, fun_name , truncated_dtype))
```



[8]:
```
# Not sure what are the units of the comoving distance? just ask:
jc.background.radial_comoving_distance?
```

Defining redshift distributions

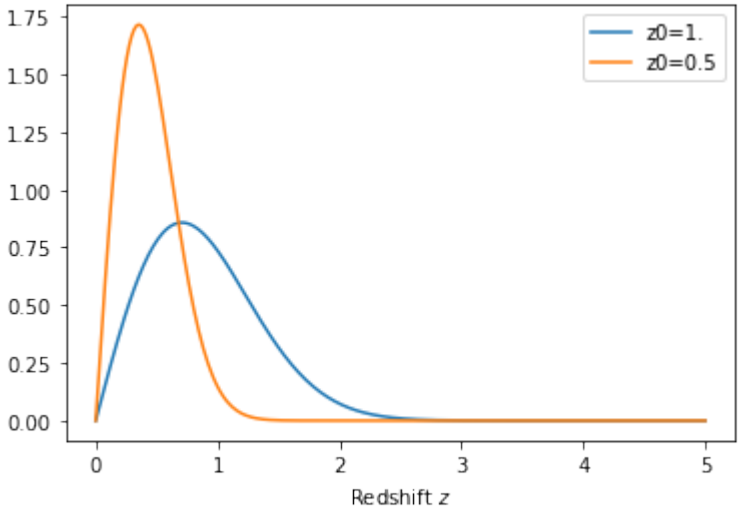
On our path to computing Fisher matrices, we need to be able to express redshift distrbutions. In `jax-cosmo` n(z) are parametrized functions which can be found in the `jax_cosmo.redshift` module.

For the purpose of this tutorial, let's see how to define a Smail type distribution:

$$n(z) = z^a \exp(-(z/z_0)^b)$$
$$which depends on 3 parameters:$$

[9]:
```
# You can inspect the documentation to see the
# meaning of these positional arguments
nz1 = jc.redshift.smail_nz(1., 2.,  1.)
nz2 = jc.redshift.smail_nz(1., 2.,  0.5)
```

[10]:
```
# And let's plot it
z = np.linspace(0,5,256)

# Redshift distributions are callable, and they return the normalized distribution
plot(z, nz1(z), label='z0=1.')
plot(z, nz2(z), label='z0=0.5')
legend();
xlabel('Redshift $z$');
```



[11]:
```
# We can check that the nz is properly normalized
jc.scipy.integrate.romb(nz1, 0., 5.)
```

[11]:
```
DeviceArray(1.0000004, dtype=float32)
```

Nice :-D

Defining probes and computing angular C_ℓ

Let's now move on to define lensing and clustering probes using these two n(z). In `jax-cosmo` a probe/tracer of a given type, i.e. lensing, contains a series of parameters, like redshift distributions, or galaxy bias. Probes are hosted in the `jax_cosmo.probes` module.

C_ℓ computations will then take as argument a list of probes and will compute all auto- and cross- correlations between all redshift bins of all probes.

First, let's define a list of redshift bins:

[12]:
```
nzs = [nz1, nz2]
```

along with 2 probes:

[13]:
```
probes = [ jc.probes.WeakLensing(nzs, sigma_e=0.26),
           jc.probes.NumberCounts(nzs, jc.bias.constant_linear_bias(1.)) ]
```

Given these probes, we can now compute tomographic angular power spectra for these probes using the `angular_cl` tools hosted in the `jax_cosmo.angular_cl` module. For now, all computations are done under the Limber approximation.

[14]:
```
ell = np.logspace(1,3) # Defines a range of \ell

# And compute the data vector
cls = jc.angular_cl.angular_cl(cosmo, ell, probes)
```

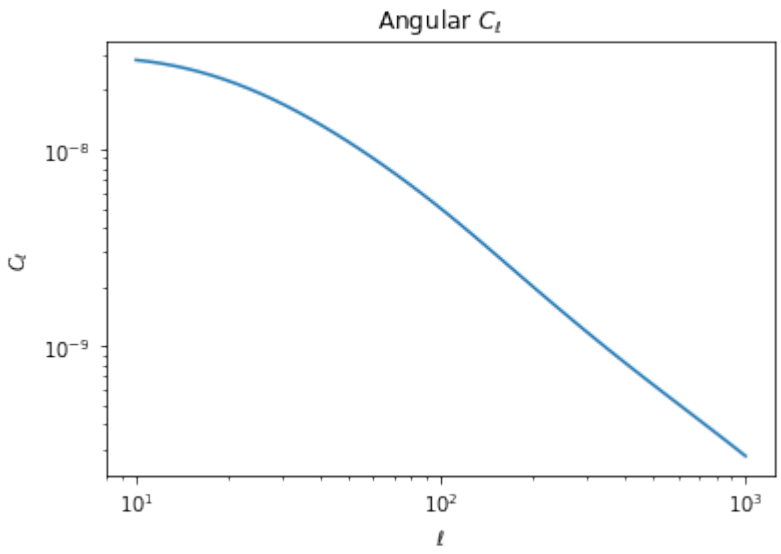
```
/usr/local/lib/python3.6/dist-packages/jax/lax/lax.py:6181: UserWarning: Explicitly␣
↪requested dtype <class 'jax.numpy.lax_numpy.int64'> requested in astype is not␣
↪available, and will be truncated to dtype int32. To enable more dtypes, set the jax_
↪enable_x64 configuration option or the JAX_ENABLE_X64 shell environment variable.␣
↪See https://github.com/google/jax#current-gotchas for more.
  warnings.warn(msg.format(dtype, fun_name , truncated_dtype))
```

[15]:
```
# Let's check the shape of these Cls
cls.shape
```

[15]:
```
(10, 50)
```

We see that we have obtained 10 spectra, each of them of size 50, which is the length of the ℓ vector. They are ordered first by probe, then by redshift bin. So the first cl is the lensing auto-spectrum of the first bin

[16]:
```
# This is for instance the first bin auto-spectrum
loglog(ell, cls[0])
ylabel(r'$C_\ell$')
xlabel(r'$\ell$');
title(r'Angular $C_\ell$');
```



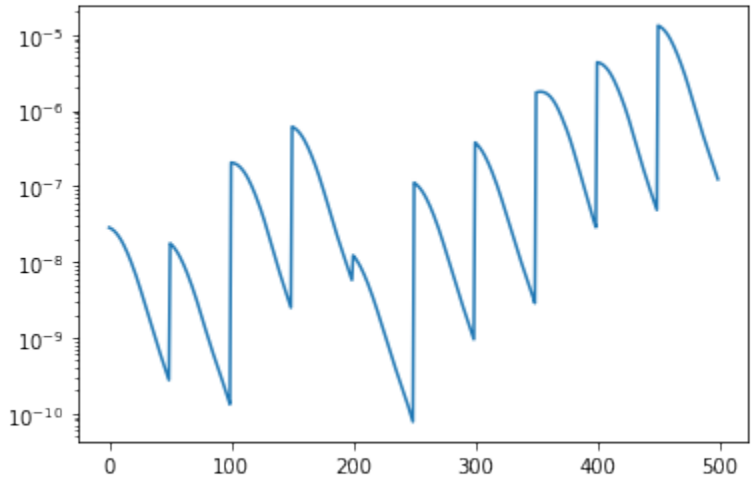
In addition to the data vector, we can also compute the covariance matrix using the tools from that module. Here is an example:

[17]:
```
mu, cov = jc.angular_cl.gaussian_cl_covariance_and_mean(cosmo, ell, probes,␣
↪sparse=True);
```

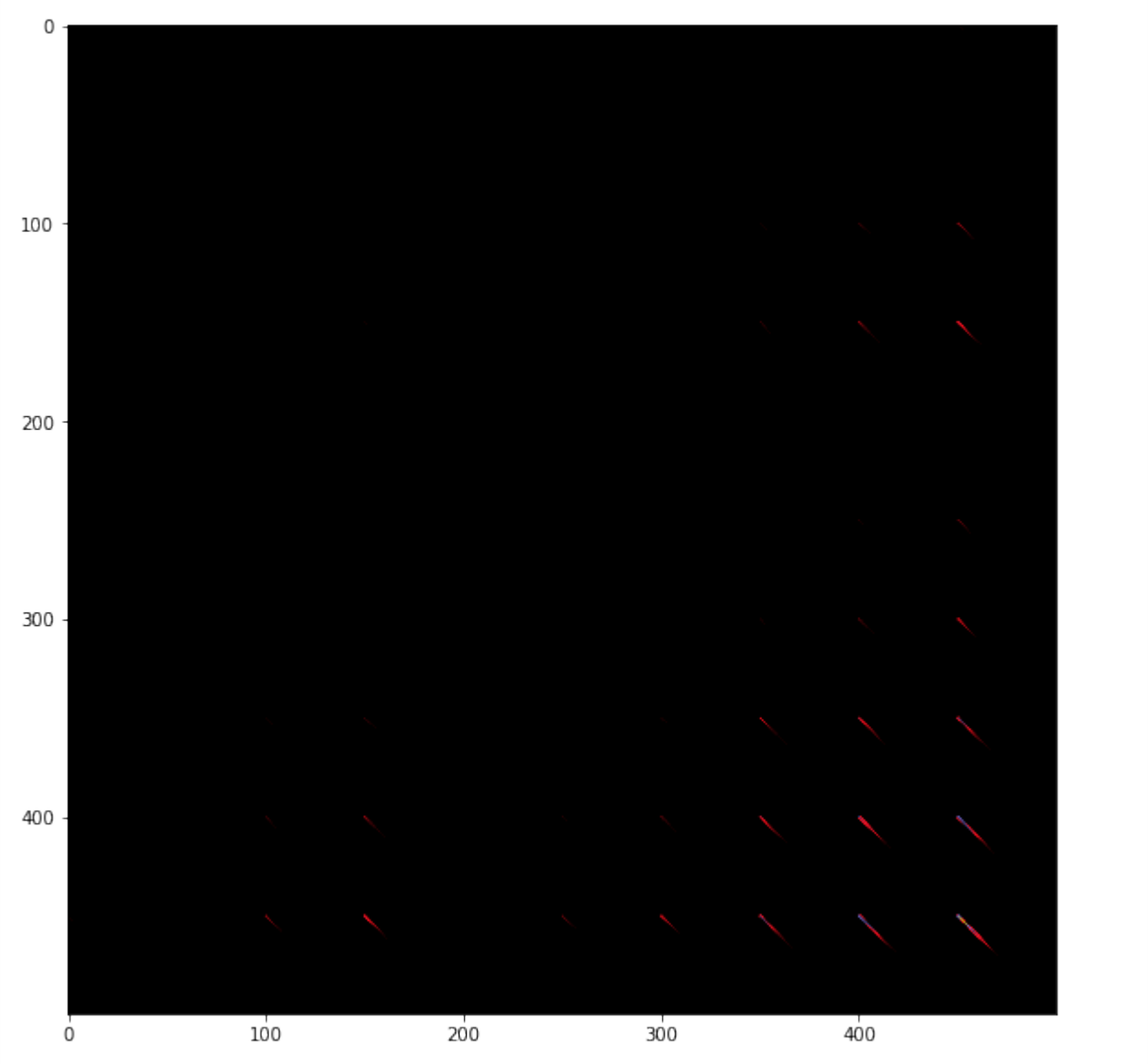
```
/usr/local/lib/python3.6/dist-packages/jax/lax/lax.py:6181: UserWarning: Explicitly␣
↪requested dtype <class 'jax.numpy.lax_numpy.int64'> requested in astype is not␣
↪available, and will be truncated to dtype int32. To enable more dtypes, set the jax_
↪enable_x64 configuration option or the JAX_ENABLE_X64 shell environment variable.␣
↪See https://github.com/google/jax#current-gotchas for more.
  warnings.warn(msg.format(dtype, fun_name , truncated_dtype))
```

The data vector from this function is in a flattened shape so that it can be multiplied by the covariance matrix easily.

[18]:
```
semilogy(mu);
```



[19]:
```
figure(figsize=(10,10))
# Here we convert the covariance matrix from sparse to dense reprensentation
# for plotting
imshow(np.log10(jc.sparse.to_dense(cov)+1e-11),cmap='gist_stern');
```



Where the wild things are: Automatic Differentiation

Now that we know how to compute various quantities, we can move on to the amazing part, computing gradients automatically by autodiff. As an example, we will demonstrate how to analytically **compute Fisher matrices, without finite differences.** But gradients are usefull for a wide range of other applications.

We begin by defining a Gaussian likelihood function for the data vector we have obtained at the previous step. And we make this likelihood function depend on an array of parameters, `Omega_c`, `sigma_8`.

[20]:
```
data = mu # We create some fake data from the fiducial cosmology
# Let's define a parameter vector for Omega_cdm, sigma8, which we initialize
# at the fiducial cosmology used to produce the data vector.
params = np.array([cosmo.Omega_c, cosmo.sigma8])

# Note the `jit` decorator for just in time compilation, this makes your code
```

(continues on next page)

(continued from previous page)

```
# run fast on GPU :-)
@jax.jit
def likelihood(p):
  # Create a new cosmology at these parameters
  cosmo = jc.Planck15(Omega_c=p[0], sigma8=p[1])

  # Compute mean and covariance of angular Cls
  m, C = jc.angular_cl.gaussian_cl_covariance_and_mean(cosmo, ell, probes,␣
↪sparse=True)

  # Return likelihood value assuming constant covariance, so we stop the gradient
  # at the level of the precision matrix, and we will not include the logdet term
  # in the likelihood
  P = jc.sparse.inv(jax.lax.stop_gradient(C))
  r = data - m
  return -0.5 * r.T @ jc.sparse.sparse_dot_vec(P, r)
```

We can try to compute the likelihood at our fiducial parameters, we should get something very close to zero:

[22]:
```
print(likelihood(params))
%timeit likelihood(params).block_until_ready()
```

```
-1.8064792e-08
10 loops, best of 3: 22.8 ms per loop
```

This is an illustration of evaluating the full likelihood. Note that because we used the `@jax.jit` decorator on the likelihood, this code is being compiled to and XLA expression that runs automatically on the GPU if it's available.

But now that we have a likelihood function of the parameters, we can manipulate it with JAX, and in particular take the second derivative of this likelihood with respect to the input cosmological parameters. This Hessian, is just minus the Fisher matrix when everything is nice and Gaussian around the fiducial comology.

So this mean, by JAX automaticatic differentiation, we can analytically derive the Fisher matrix in just one line:

[23]:
```
# Compile a function that computes the Hessian of the likelihood
hessian_loglik = jax.jit(jax.hessian(likelihood))

# Evalauate the Hessian at fiductial cosmology to retrieve Fisher matrix
# This is a bit slow at first....
F = - hessian_loglik(params)
```

```
/usr/local/lib/python3.6/dist-packages/jax/lax/lax.py:6181: UserWarning: Explicitly␣
↪requested dtype <class 'jax.numpy.lax_numpy.int64'> requested in astype is not␣
↪available, and will be truncated to dtype int32. To enable more dtypes, set the jax_
↪enable_x64 configuration option or the JAX_ENABLE_X64 shell environment variable.␣
↪See https://github.com/google/jax#current-gotchas for more.
  warnings.warn(msg.format(dtype, fun_name , truncated_dtype))
```

What we are doing on the line above is taking the Hessian of the likelihood function, and evaluating at the fiducial cosmology. We surround the whole thing with a `jit` instruction so that the function gets compiled and evaluated in one block in the GPU.

Compiling the function is not instantaneous, but **once compiled it becomes fast**:

[25]:
```
%timeit hessian_loglik(params).block_until_ready()
```

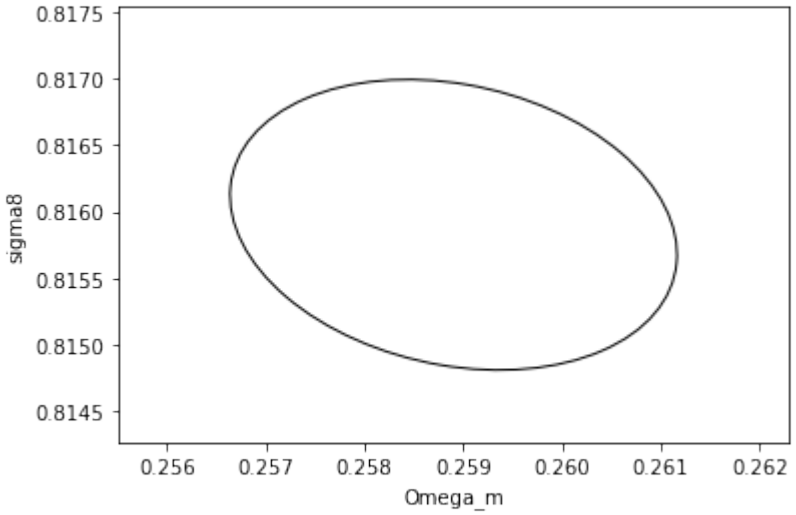
```
1 loop, best of 3: 302 ms per loop
```

And best of all: **No derivatives were harmed by finite differences in the computation of this Fisher!**

We can now try to plot it:

[26]:
```
# We can now plot contours obtained with this
plot_contours(F, params, fill=False);
xlabel('Omega_m')
ylabel('sigma8')
```

[26]:
```
Text(8.125, 0.5, 'sigma8')
```



And just to reinforce this point and demonstrate further audodiff magic, let's try to derive the same matrix differently, using the usual formula for constant covariance:

$$F_{\alpha,\beta} = \sum_{i,j} \frac{d\mu_i}{d\theta_\alpha} C^{-1}_{i,j} \frac{d\mu_j}{d\theta_\beta}$$

What we need in this expression, is the covariance matrix, which we already have and the Jacobian of the mean with respect to parameters. Normally you would need to use finite differencing, but luckily we can get that easily with JAX:

[27]:
```
# We define a parameter dependent function that computes the mean
def mean_fn(p):
  cosmo = jc.Planck15(Omega_c=p[0], sigma8=p[1])
  # Compute signal vector
  m = jc.angular_cl.angular_cl(cosmo, ell, probes)
  return m.flatten() # We want it in 1d to operate against the covariance matrix
```

[28]:
```
# We compute it's jacobian with JAX, and we JIT it for efficiency
jac_mean = jax.jit(jax.jacfwd(mean_fn))
```

[29]:
```
# We can now evaluate the jacobian at the fiducial cosmology
dmu = jac_mean(params)
```

```
/usr/local/lib/python3.6/dist-packages/jax/lax/lax.py:6181: UserWarning: Explicitly␣
↪requested dtype <class 'jax.numpy.lax_numpy.int64'> requested in astype is not␣
↪available, and will be truncated to dtype int32. To enable more dtypes, set the jax_
↪enable_x64 configuration option or the JAX_ENABLE_X64 shell environment variable.␣
↪See https://github.com/google/jax#current-gotchas for more.
  warnings.warn(msg.format(dtype, fun_name , truncated_dtype))
```

[30]:
```
dmu.shape
```

[30]:
```
(500, 2)
```

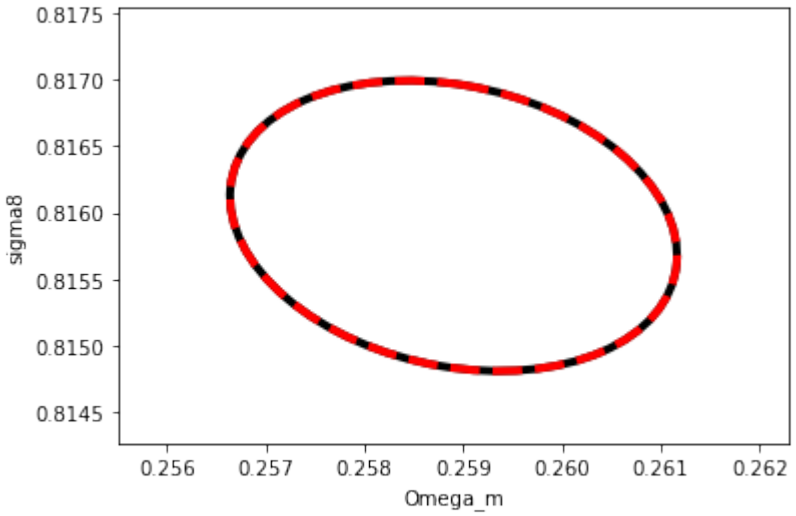
[31]:
```
# For fun, we can also time it
%timeit jac_mean(params).block_until_ready()
```

```
10 loops, best of 3: 62 ms per loop
```

Getting these gradients is the same order of time than evaluating the forward function!

[32]:
```
# Now we can compose the Fisher matrix:
F_2 = jc.sparse.dot(dmu.T, jc.sparse.inv(cov), dmu)
```

[33]:
```
# We can now plot contours obtained with this
plot_contours(F, params, fill=False,color='black',lw=4);
plot_contours(F_2, params, fill=False, color='red', lw=4, linestyle='dashed');
xlabel('Omega_m')
ylabel('sigma8');
```



The red dashed is our second derivation of the Fisher matrix using the jacobian, the black contour underneath is our first derivation simply taking the Hessian of the likelihood.

They agree perfectly, and they should, because they are both analytically computed.

Conclusions and going further

We have covered some of the most important points of `jax-cosmo`, feel free to go through the design document for background and further explanations of how things work. You can also follow this JAX document to go deeper into JAX.

`jax-cosmo` is still very young and lacks many features, but hopefuly this notebook demonstrates the power of automatic differentiation, and given that the entire code is in simple Python, feel free to contribute missing features that would be necessary for your work ;-)

1.2 Comparison to CCL

This notebook compares the implementation from jax_cosmo to CCL

[1]:
```
%pylab inline
import os
os.environ['JAX_ENABLE_X64']='True'

import pyccl as ccl
import jax
from jax_cosmo import Cosmology, background
```

```
Populating the interactive namespace from numpy and matplotlib
```

[2]:
```
# We first define equivalent CCL and jax_cosmo cosmologies
cosmo_ccl = ccl.Cosmology(
    Omega_c=0.3, Omega_b=0.05, h=0.7, sigma8 = 0.8, n_s=0.96, Neff=0,
    transfer_function='eisenstein_hu', matter_power_spectrum='halofit')

cosmo_jax = Cosmology(Omega_c=0.3, Omega_b=0.05, h=0.7, sigma8 = 0.8, n_s=0.96,
                      Omega_k=0., w0=-1., wa=0.)
```

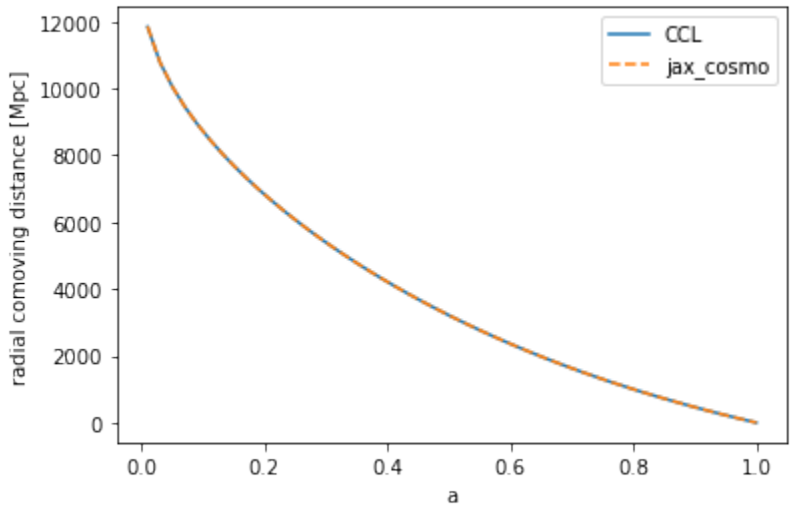
1.2.1 Comparing background cosmology

[3]:
```
# Test array of scale factors
a = np.linspace(0.01, 1.)
```

[4]:
```
# Testing the radial comoving distance
chi_ccl = ccl.comoving_radial_distance(cosmo_ccl, a)
chi_jax = background.radial_comoving_distance(cosmo_jax, a)/cosmo_jax.h

plot(a, chi_ccl, label='CCL')
plot(a, chi_jax, '--', label='jax_cosmo')
legend()
xlabel('a')
ylabel('radial comoving distance [Mpc]')
```

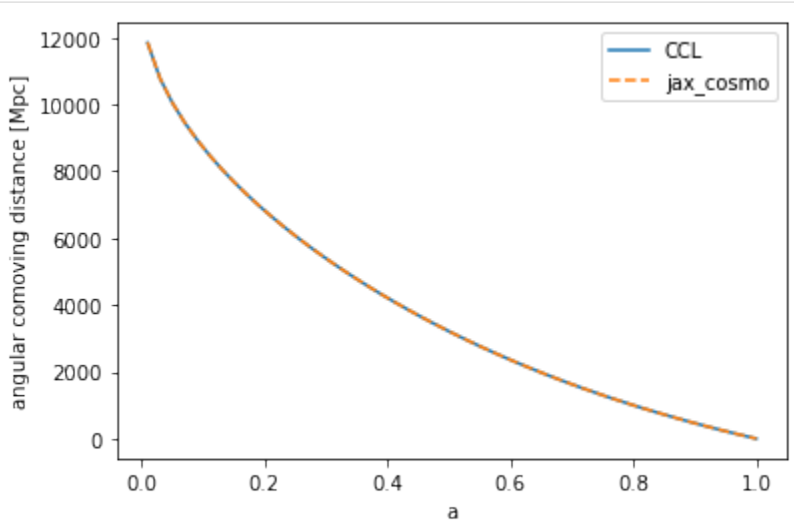
```
/home/francois/.local/lib/python3.8/site-packages/jax/lib/xla_bridge.py:116:␣
↪UserWarning: No GPU/TPU found, falling back to CPU.
  warnings.warn('No GPU/TPU found, falling back to CPU.')
```

[4]:
```
Text(0, 0.5, 'radial comoving distance [Mpc]')
```



[5]:
```
# Testing the angular comoving distance
chi_ccl = ccl.comoving_angular_distance(cosmo_ccl, a)
chi_jax = background.transverse_comoving_distance(cosmo_jax, a)/cosmo_jax.h

plot(a, chi_ccl, label='CCL')
plot(a, chi_jax, '--', label='jax_cosmo')
legend()
xlabel('a')
ylabel('angular comoving distance [Mpc]')
```

[5]:
```
Text(0, 0.5, 'angular comoving distance [Mpc]')
```



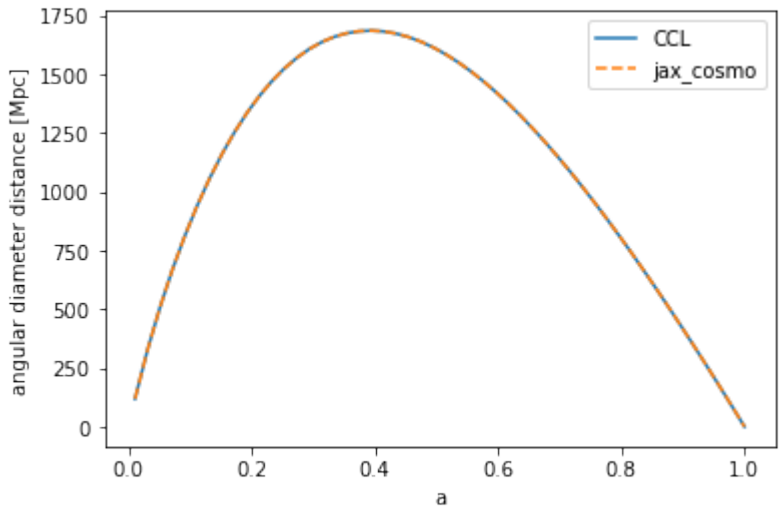
[6]:
```
# Testing the angular diameter distance
chi_ccl = ccl.angular_diameter_distance(cosmo_ccl, a)
chi_jax = background.angular_diameter_distance(cosmo_jax, a)/cosmo_jax.h

plot(a, chi_ccl, label='CCL')
plot(a, chi_jax, '--', label='jax_cosmo')
legend()
xlabel('a')
```

(continues on next page)

(continued from previous page)

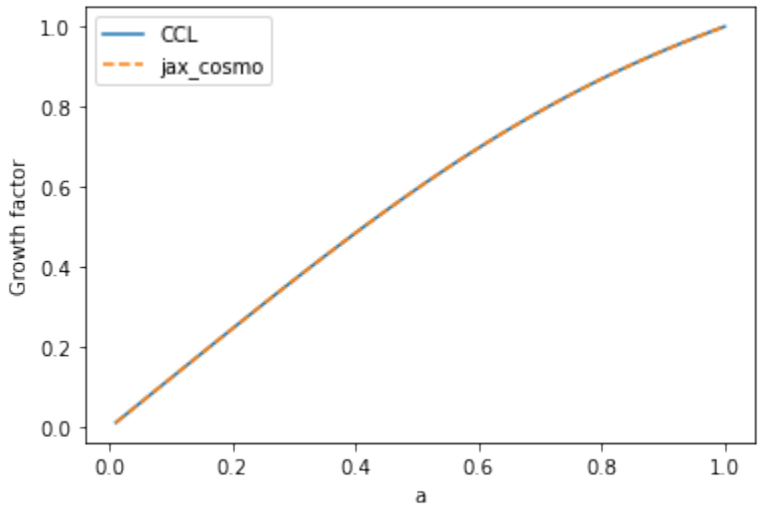
```
ylabel('angular diameter distance [Mpc]')
```

[6]:
```
Text(0, 0.5, 'angular diameter distance [Mpc]')
```



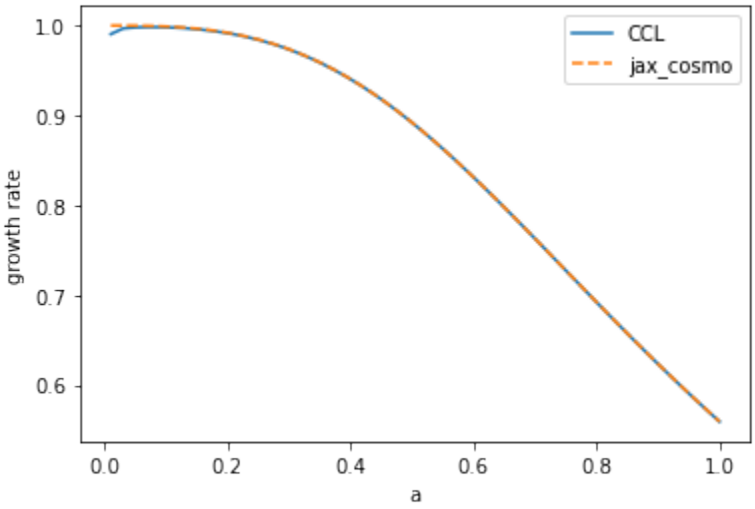
[7]:
```
# Comparing the growth factor
plot(a, ccl.growth_factor(cosmo_ccl,a), label='CCL')
plot(a, background.growth_factor(cosmo_jax, a), '--', label='jax_cosmo')
legend()
xlabel('a')
ylabel('Growth factor')
```

[7]:
```
Text(0, 0.5, 'Growth factor')
```



[8]:
```
# Comparing linear growth rate
plot(a, ccl.growth_rate(cosmo_ccl,a), label='CCL')
plot(a, background.growth_rate(cosmo_jax, a), '--', label='jax_cosmo')
legend()
xlabel('a')
ylabel('growth rate')
```

[8]:
```
Text(0, 0.5, 'growth rate')
```

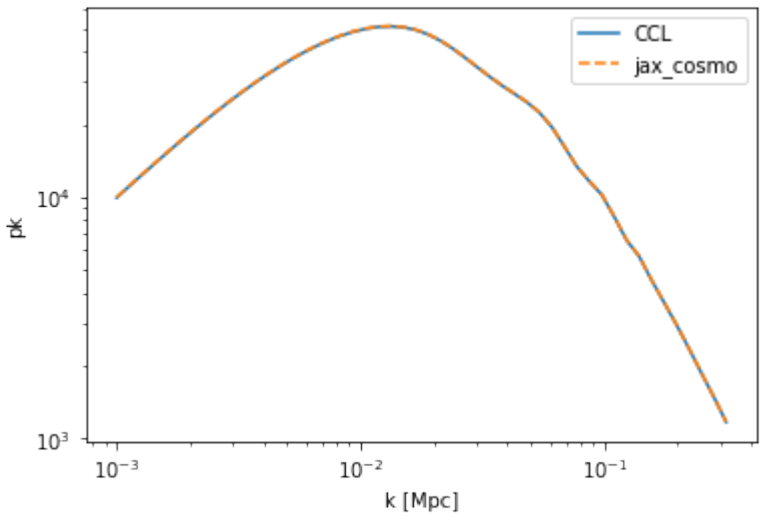


1.2.2 Comparing matter power spectrum

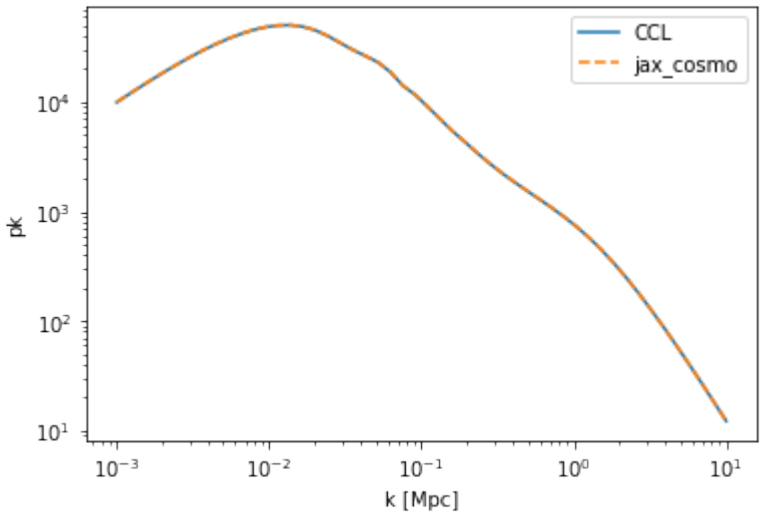
[9]:
```
from jax_cosmo.power import linear_matter_power, nonlinear_matter_power
```

[9]:
```
k = np.logspace(-3,-0.5)
```

[10]:
```
#Let's have a look at the linear power
pk_ccl = ccl.linear_matter_power(cosmo_ccl, k, 1.0)
pk_jax = linear_matter_power(cosmo_jax, k/cosmo_jax.h, a=1.0)

loglog(k,pk_ccl,label='CCL')
loglog(k,pk_jax/cosmo_jax.h**3, '--', label='jax_cosmo')
legend()
xlabel('k [Mpc]')
ylabel('pk');
```



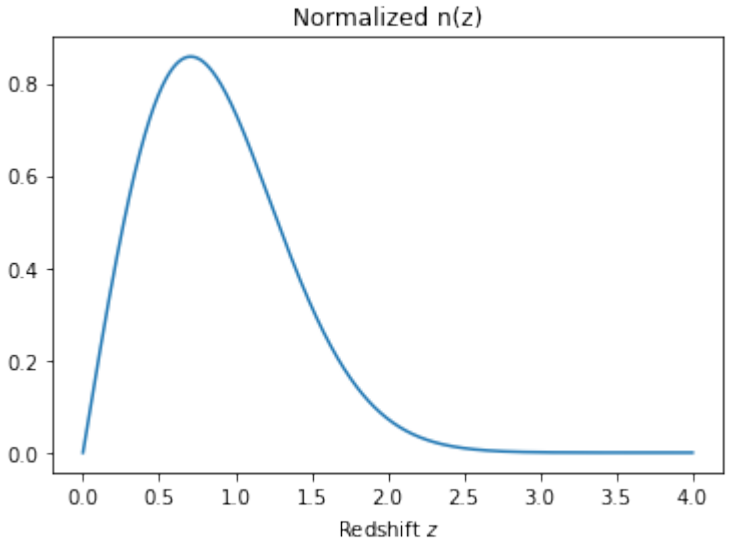
[11]:
```
k = np.logspace(-3,1)
#Let's have a look at the non linear power
pk_ccl = ccl.nonlin_matter_power(cosmo_ccl, k, 1.0)
pk_jax = nonlinear_matter_power(cosmo_jax, k/cosmo_jax.h, a=1.0)

loglog(k,pk_ccl,label='CCL')
loglog(k,pk_jax/cosmo_jax.h**3, '--', label='jax_cosmo')
legend()
xlabel('k [Mpc]')
ylabel('pk');
```



1.2.3 Comparing angular cl

[12]:
```
from jax_cosmo.redshift import smail_nz

# Let's define a redshift distribution
# with a Smail distribution with a=1, b=2, z0=1
nz = smail_nz(1.,2., 1.)
```

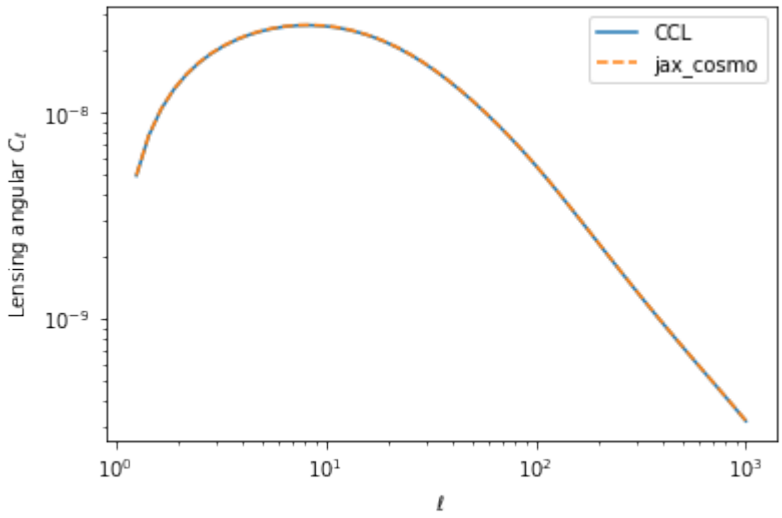
[13]:
```
z = linspace(0,4,1024)
plot(z, nz(z))
xlabel(r'Redshift $z$');
title('Normalized n(z)');
```



[14]:
```
from jax_cosmo.angular_cl import angular_cl
from jax_cosmo import probes

# Let's first compute some Weak Lensing cls
tracer_ccl = ccl.WeakLensingTracer(cosmo_ccl, (z, nz(z)), use_A_ia=False)
tracer_jax = probes.WeakLensing([nz])

ell = np.logspace(0.1,3)

cl_ccl = ccl.angular_cl(cosmo_ccl, tracer_ccl, tracer_ccl, ell)
cl_jax = angular_cl(cosmo_jax, ell, [tracer_jax])
```

[15]:
```
loglog(ell, cl_ccl,label='CCL')
loglog(ell, cl_jax[0], '--', label='jax_cosmo')

legend()
xlabel(r'$\ell$')
ylabel(r'Lensing angular $C_\ell$')
```

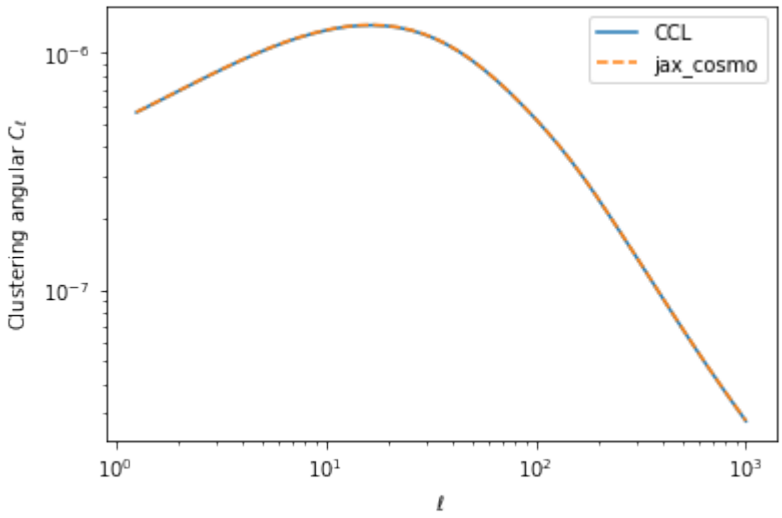
[15]:
```
Text(0, 0.5, 'Lensing angular $C_\\ell$')
```



[16]:
```
# Let's try galaxy clustering now
from jax_cosmo.bias import constant_linear_bias

# We define a trivial bias model
bias = constant_linear_bias(1.)

tracer_ccl_n = ccl.NumberCountsTracer(cosmo_ccl,
                                      has_rsd=False,
                                      dndz=(z, nz(z)),
                                      bias=(z, bias(cosmo_jax, z)))
tracer_jax_n = probes.NumberCounts([nz], bias)

cl_ccl = ccl.angular_cl(cosmo_ccl, tracer_ccl_n, tracer_ccl_n, ell)
cl_jax = angular_cl(cosmo_jax, ell, [tracer_jax_n])
```

[17]:
```
import jax_cosmo.constants as cst
loglog(ell, cl_ccl,label='CCL')
loglog(ell, cl_jax[0], '--', label='jax_cosmo')

legend()
xlabel(r'$\ell$')
ylabel(r'Clustering angular $C_\ell$')
```

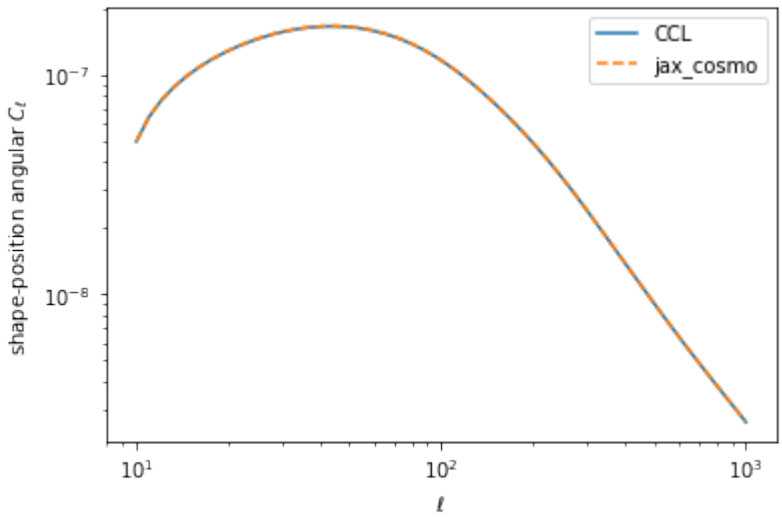
[17]:
```
Text(0, 0.5, 'Clustering angular $C_\\ell$')
```



[18]:
```
# And  finally.... a cross-spectrum

cl_ccl = ccl.angular_cl(cosmo_ccl, tracer_ccl, tracer_ccl_n, ell)
cl_jax = angular_cl(cosmo_jax, ell, [tracer_jax, tracer_jax_n])
```

[19]:
```
ell = np.logspace(1,3)

loglog(ell, cl_ccl,label='CCL')
loglog(ell, cl_jax[1], '--', label='jax_cosmo')

legend()
xlabel(r'$\ell$')
ylabel(r'shape-position angular $C_\ell$')
```

[19]:
```
Text(0, 0.5, 'shape-position angular $C_\\ell$')
```



[ ]:

1.3 jax_cosmo public API

1.3.1 jax_cosmo package

jax_cosmo.angular_cl module

jax_cosmo.angular_cl.**angular_cl**(*cosmo*, *ell*, *probes*, *transfer_fn=<function Eisenstein_Hu>*, *nonlinear_fn=<function halofit>*)

Computes angular Cls for the provided probes

All using the Limber approximation

Returns **cls**

Return type [ell, ncls]

jax_cosmo.angular_cl.**gaussian_cl_covariance**(*ell*, *probes*, *cl_signal*, *cl_noise*, *f_sky=0.25*, *sparse=True*)

Computes a Gaussian covariance for the angular cls of the provided probes

Set sparse True to return a sparse matrix representation that uses a factor of n_ell less memory and is compatible with the linear algebra operations in `jax_cosmo.sparse`.

return_cls: (returns covariance)

jax_cosmo.angular_cl.**gaussian_cl_covariance_and_mean**(*cosmo*, *ell*, *probes*, *transfer_fn=<function Eisenstein_Hu>*, *nonlinear_fn=<function halofit>*, *f_sky=0.25*, *sparse=False*)

Computes a Gaussian covariance for the angular cls of the provided probes

Set sparse True to return a sparse matrix representation that uses a factor of n_ell less memory and is compatible with the linear algebra operations in `jax_cosmo.sparse`.

return_cls: (returns signal + noise cl, covariance)

jax_cosmo.angular_cl.**noise_cl**(*ell*, *probes*)

Computes noise contributions to auto-spectra

jax_cosmo.background module

jax_cosmo.background.**w**(*cosmo*, *a*)

Dark Energy equation of state parameter using the Linder parametrisation.

Parameters

- **cosmo** (*Cosmology*) – Cosmological parameters structure
- **a** (*array_like*) – Scale factor

Returns **w** – The Dark Energy equation of state parameter at the specified scale factor

Return type ndarray, or float if input scalar

Notes

The Linder parametrization **:cite:‘2003:Linder‘** for the Dark Energy equation of state $p = w\rho$ is given by:

$$w(a) = w_0 + w_a(1 - a)$$

jax_cosmo.background.**f_de**(*cosmo*, *a*)
Evolution parameter for the Dark Energy density.

Parameters **a** (*array_like*) – Scale factor

Returns **f** – The evolution parameter of the Dark Energy density as a function of scale factor

Return type ndarray, or float if input scalar

Notes

For a given parametrisation of the Dark Energy equation of state, the scaling of the Dark Energy density with time can be written as:

$$\rho_{de}(a) = \rho_{de}(a = 1)e^{f(a)}$$

(see **:cite:‘2005:Percival‘** and note the difference in the exponent base in the parametrizations) where $f(a)$ is computed as $f(a) = -3\int_0^{\ln(a)}[1 + w(a')]d\ln(a')$. In the case of Linder's parametrisation for the dark energy in Eq. `linderParam` $f(a)$ becomes:

$$f(a) = -3(1 + w_0 + w_a)\ln(a) + 3w_a(a - 1)$$

jax_cosmo.background.**Esqr**(*cosmo*, *a*)
Square of the scale factor dependent factor E(a) in the Hubble parameter.

Parameters **a** (*array_like*) – Scale factor

Returns **E^2** – Square of the scaling of the Hubble constant as a function of scale factor

Return type ndarray, or float if input scalar

Notes

The Hubble parameter at scale factor a is given by $H^2(a) = E^2(a)H_o^2$ where E^2 is obtained through Friedman's Equation (see **:cite:‘2005:Percival‘**) :

$$E^2(a) = \Omega_m a^{-3} + \Omega_k a^{-2} + \Omega_{de} e^{f(a)}$$

where $f(a)$ is the Dark Energy evolution parameter computed by `f_de()`.

jax_cosmo.background.**H**(*cosmo*, *a*)
Hubble parameter [km/s/(Mpc/h)] at scale factor a

Parameters **a** (*array_like*) – Scale factor

Returns **H** – Hubble parameter at the requested scale factor.

Return type ndarray, or float if input scalar

jax_cosmo.background.**Omega_m_a**(*cosmo*, *a*)
Matter density at scale factor a.

Parameters **a** (*array_like*) – Scale factor

Returns **Omega_m** – Non-relativistic matter density at the requested scale factor

Return type ndarray, or float if input scalar

Notes

The evolution of matter density $\Omega_m(a)$ is given by:

$$\Omega_m(a) = \frac{\Omega_m a^{-3}}{E^2(a)}$$

see **:cite:‘2005:Percival‘** Eq. (6)

jax_cosmo.background.**Omega_de_a**(*cosmo*, *a*)
Dark Energy density at scale factor a.

Parameters **a** (*array_like*) – Scale factor

Returns **Omega_de** – Dark Energy density at the requested scale factor

Return type ndarray, or float if input scalar

Notes

The evolution of Dark Energy density $\Omega_{de}(a)$ is given by:

$$\Omega_{de}(a) = \frac{\Omega_{de} e^{f(a)}}{E^2(a)}$$

where $f(a)$ is the Dark Energy evolution parameter computed by `f_de()` (see **:cite:‘2005:Percival‘** Eq. (6)).

jax_cosmo.background.**radial_comoving_distance**(*cosmo*, *a*, *log10_amin=-3*, *steps=256*)
Radial comoving distance in [Mpc/h] for a given scale factor.

Parameters **a** (*array_like*) – Scale factor

Returns **chi** – Radial comoving distance corresponding to the specified scale factor.

Return type ndarray, or float if input scalar

Notes

The radial comoving distance is computed by performing the following integration:

$$\chi(a) = R_H \int_a^1 \frac{da'}{a'^2 E(a')}$$

jax_cosmo.background.**dchioverda**(*cosmo*, *a*)
Derivative of the radial comoving distance with respect to the scale factor.

Parameters **a** (*array_like*) – Scale factor

Returns **dchi/da** – Derivative of the radial comoving distance with respect to the scale factor at the specified scale factor.

Return type ndarray, or float if input scalar

Notes

The expression for $\frac{d\chi}{da}$ is:

$$\frac{d\chi}{da}(a) = \frac{R_H}{a^2 E(a)}$$

jax_cosmo.background.**transverse_comoving_distance**(*cosmo*, *a*)
Transverse comoving distance in [Mpc/h] for a given scale factor.

Parameters **a** (*array_like*) – Scale factor

Returns **f_k** – Transverse comoving distance corresponding to the specified scale factor.

Return type ndarray, or float if input scalar

Notes

The transverse comoving distance depends on the curvature of the universe and is related to the radial comoving distance through:

$$f_k(a) = \begin{cases} R_H \frac{1}{\sqrt{\Omega_k}} \sinh(\sqrt{|\Omega_k|}\chi(a) R_H) & \text{for } \Omega_k > 0 \\ \chi(a) & \text{for } \Omega_k = 0 \\ R_H \frac{1}{\sqrt{\Omega_k}} \sin(\sqrt{|\Omega_k|}\chi(a) R_H) & \text{for } \Omega_k < 0 \end{cases}$$

jax_cosmo.background.**angular_diameter_distance**(*cosmo*, *a*)
Angular diameter distance in [Mpc/h] for a given scale factor.

Parameters **a** (*array_like*) – Scale factor

Returns **d_A**

Return type ndarray, or float if input scalar

Notes

Angular diameter distance is expressed in terms of the transverse comoving distance as:

$$d_A(a) = a f_k(a)$$

jax_cosmo.background.**growth_factor**(*cosmo*, *a*)
Compute linear growth factor D(a) at a given scale factor, normalized such that D(a=1) = 1.

Parameters

- **cosmo** (*Cosmology*) – Cosmology object
- **a** (*array_like*) – Scale factor

Returns **D** – Growth factor computed at requested scale factor

Return type ndarray, or float if input scalar

Notes

The growth computation will depend on the cosmology parametrization, for instance if the $gamma$ parameter is defined, the growth will be computed assuming the $f = Omega^gamma$ growth rate, otherwise the usual ODE for growth will be solved.

jax_cosmo.background.**growth_rate**(*cosmo*, *a*)
Compute growth rate dD/dlna at a given scale factor.

Parameters

- **cosmo** (*Cosmology*) – Cosmology object
- **a** (*array_like*) – Scale factor

Returns **f** – Growth rate computed at requested scale factor

Return type ndarray, or float if input scalar

Notes

The growth computation will depend on the cosmology parametrization, for instance if the $gamma$ parameter is defined, the growth will be computed assuming the $f = Omega^gamma$ growth rate, otherwise the usual ODE for growth will be solved.

The LCDM approximation to the growth rate $f_\gamma(a)$ is given by:

$$f_\gamma(a) = \Omega_m^\gamma(a)$$

$$with : math : `\gamma` in LCDM, given approximately by : .. math ::$$

$$\gamma = 0.55$$

see **:cite:‘2019:Euclid Preparation VII, eqn.32‘**

jax_cosmo.bias module

class jax_cosmo.bias.**constant_linear_bias**(**args*, ***kwargs*)
Bases: *jax_cosmo.jax_utils.container*

Class representing a linear bias

b: redshift independent bias value

class jax_cosmo.bias.**des_y1_ia_bias**(**args*, ***kwargs*)
Bases: *jax_cosmo.jax_utils.container*

https://arxiv.org/pdf/1708.01538.pdf Sec. VII.B

cosmo: cosmology A: amplitude eta: redshift dependent slope z0: pivot redshift

class jax_cosmo.bias.**inverse_growth_linear_bias**(**args*, ***kwargs*)
Bases: *jax_cosmo.jax_utils.container*

TODO: what’s a better name for this? Class representing an inverse bias in 1/growth(a)

cosmo: cosmology b: redshift independent bias value at z=0

jax_cosmo.constants module

Created on Jun 12, 2013 @author: Francois Lanusse <francois.lanusse@cea.fr>

C_1: Instrinsic alignment normalisation constant [(h^2 M_sun Mpc^{-3})^{-1}], see Kirk et al 2010. NB: Bridle & King report different units, but is a typo. c : Speed of light in [km/s] eta_nu: ratio of energy density in neutrinos to energy in photons h0: Hubble constant in [km/s/(h^{-1} Mpc)] rh : Hubble radius in [h^{-1} Mpc] rho_crit: Critical density of Universe in units of [h^2 M_sun Mpc^{-3}]. tcmb : Temperature of the CMB today in [K]

jax_cosmo.jax_utils module

class jax_cosmo.jax_utils.**container**(**args*, ***kwargs*)
Bases: object

Generic structure to trace a parameterized function

Paramters for the object, i.e. things that need to be traced for autodiff are stored as a list in self.params Configuration arguments, i.e. static things that do not need to be traced are stored as a dictionary in self.config This is for things like flags or type of PS or things like that.

tree_flatten()

classmethod tree_unflatten(*aux_data*, *children*)

jax_cosmo.power module

jax_cosmo.power.**primordial_matter_power**(*cosmo*, *k*)
Primordial power spectrum Pk = k^n

jax_cosmo.power.**linear_matter_power**(*cosmo*, *k*, *a=1.0*, *transfer_fn=<function Eisenstein_Hu>*, ***kwargs*)
Computes the linear matter power spectrum.

Parameters

- **k** (*array_like*) – Wave number in h Mpc^{-1}
- **a** (*array_like, optional*) – Scale factor (def: 1.0)
- **transfer_fn** (transfer_fn(cosmo, k, **kwargs)) – Transfer function

Returns **pk** – Linear matter power spectrum at the specified scale and scale factor.

Return type array_like

jax_cosmo.power.**nonlinear_matter_power**(*cosmo*, *k*, *a=1.0*, *transfer_fn=<function Eisenstein_Hu>*, *nonlinear_fn=<function halofit>*)
Computes the non-linear matter power spectrum.

This function is just a wrapper over several nonlinear power spectra.

jax_cosmo.probes module

class jax_cosmo.probes.**WeakLensing**(*redshift_bins*, *ia_bias=None*, *multiplicative_bias=0.0*, *sigma_e=0.26*, ***kwargs*)
Bases: *jax_cosmo.jax_utils.container*

Class representing a weak lensing probe, with a bunch of bins

redshift_bins: list of nzredshift distributions ia_bias: (optional) if provided, IA will be added with the NLA model, either a single bias object or a list of same size as nzs multiplicative_bias: (optional) adds an (1+m) multiplicative bias, either single value or list of same length as redshift bins

sigma_e: intrinsic galaxy ellipticity

kernel(*cosmo*, *z*, *ell*)
Compute the radial kernel for all nz bins in this probe.

radial_kernel: shape (nbins, nz)

n_tracers
Returns the number of tracers for this probe, i.e. redshift bins

noise()
Returns the noise power for all redshifts return: shape [nbins]

zmax
Returns the maximum redsfhit probed by this probe

class jax_cosmo.probes.**NumberCounts**(*redshift_bins*, *bias*, *has_rsd=False*, ***kwargs*)
Bases: *jax_cosmo.jax_utils.container*

Class representing a galaxy clustering probe, with a bunch of bins

redshift_bins: nzredshift distributions

has_rsd. . . .

kernel(*cosmo*, *z*, *ell*)
Compute the radial kernel for all nz bins in this probe.

radial_kernel: shape (nbins, nz)

n_tracers
Returns the number of tracers for this probe, i.e. redshift bins

noise()
Returns the noise power for all redshifts return: shape [nbins]

zmax
Returns the maximum redsfhit probed by this probe

jax_cosmo.redshift module

class jax_cosmo.redshift.**smail_nz**(**args*, *gals_per_arcmin2=1.0*, *zmax=10.0*, ***kwargs*)
Bases: jax_cosmo.redshift.redshift_distribution

Defines a smail distribution with these arguments Parameters: ———— a:

b:

z0:

gals_per_arcmin2: number of galaxies per sq arcmin

pz_fn(*z*)
Un-normalized n(z) function provided by sub classes

class jax_cosmo.redshift.**kde_nz**(**args*, *gals_per_arcmin2=1.0*, *zmax=10.0*, ***kwargs*)
Bases: jax_cosmo.redshift.redshift_distribution

A redshift distribution based on a KDE estimate of the nz of a given catalog currently uses a Gaussian kernel. TODO: add more if necessary

zcat: redshift catalog weights: weight for each galaxy between 0 and 1

bw: Bandwidth for the KDE

Example: nz = kde_nz(redshift_catalog, w, bw=0.1)

pz_fn(*z*)
Un-normalized n(z) function provided by sub classes

class jax_cosmo.redshift.**delta_nz**(**args*, ***kwargs*)
Bases: jax_cosmo.redshift.redshift_distribution

Defines a single plane redshift distribution with these arguments Parameters: ———— z0:

pz_fn(*z*)
Un-normalized n(z) function provided by sub classes

jax_cosmo.transfer module

jax_cosmo.transfer.**Eisenstein_Hu**(*cosmo*, *k*, *type='eisenhu_osc'*)
Computes the Eisenstein & Hu matter transfer function.

Parameters

- **cosmo** (*Background*) – Background cosmology
- **k** (*array_like*) – Wave number in h Mpc^{-1}
- **type** (*str, optional*) – Type of transfer function. Either 'eisenhu' or 'eisenhu_osc' (def: 'eisenhu_osc')

Returns **T** – Value of the transfer function at the requested wave number

Return type array_like

Notes

The Eisenstein & Hu transfer functions are computed using the fitting formulae of **:cite:'1998:EisensteinHu'**

jax_cosmo.utils module

jax_cosmo.utils.**a2z**(*a*)
converts from scale factor to redshift

jax_cosmo.utils.**z2a**(*z*)
converts from redshift to scale factor

1.3.2 jax_cosmo.scipy package

jax_cosmo.scipy.integrate module

jax_cosmo.scipy.integrate.**romb**(*function*, *a*, *b*, *args=()*, *divmax=6*, *return_error=False*)
Romberg integration of a callable function or method. Returns the integral of *function* (a function of one variable) over the interval (*a*, *b*). If *show* is 1, the triangular array of the intermediate results will be printed. If *vec_func* is True (default is False), then *function* is assumed to support vector arguments. :param function: Function to be integrated. :type function: callable :param a: Lower limit of integration. :type a: float :param b: Upper limit of integration. :type b: float

Returns **results** – Result of the integration.

Return type float

Other Parameters

- **args** (*tuple, optional*) – Extra arguments to pass to function. Each element of *args* will be passed as a single argument to *func*. Default is to pass no extra arguments.
- **divmax** (*int, optional*) – Maximum order of extrapolation. Default is 10.

See also:

fixed_quad() Fixed-order Gaussian quadrature.

quad() Adaptive quadrature using QUADPACK.

dblquad() Double integrals.

tplquad() Triple integrals.

romb() Integrators for sampled data.

simps() Integrators for sampled data.

cumtrapz() Cumulative integration for sampled data.

ode() ODE integrator.

odeint() ODE integrator.

References

Examples

Integrate a gaussian from 0 to 1 and compare to the error function. >>> from scipy import integrate >>> from scipy.special import erf >>> gaussian = lambda x: 1/np.sqrt(np.pi) * np.exp(-x**2) >>> result = integrate.romberg(gaussian, 0, 1, show=True) Romberg integration of <function vfunc at ...> from [0, 1]

```
Steps  StepSize  Results
    1  1.000000  0.385872
    2  0.500000  0.412631  0.421551
    4  0.250000  0.419184  0.421368  0.421356
    8  0.125000  0.420810  0.421352  0.421350  0.421350
   16  0.062500  0.421215  0.421350  0.421350  0.421350  0.421350
   32  0.031250  0.421317  0.421350  0.421350  0.421350  0.421350  0.421350
```

The final result is 0.421350396475 after 33 function evaluations. >>> print("%g %g" % (2*result, erf(1))) 0.842701 0.842701

jax_cosmo.scipy.integrate.**simps**(*f*, *a*, *b*, *N=128*)

Approximate the integral of f(x) from a to b by Simpson's rule.

Simpson's rule approximates the integral int_a^b f(x) dx by the sum: (dx/3) sum_{k=1}^{N/2} (f(x_{2i-2} + 4f(x_{2i-1}) + f(x_{2i})) where x_i = a + i*dx and dx = (b - a)/N.

Parameters

- **f** (*function*) – Vectorized function of a single variable
- **, b** (*a*) – Interval of integration [a,b]
- **N** (*(even) integer*) – Number of subintervals of [a,b]

Returns Approximation of the integral of f(x) from a to b using Simpson's rule with N subintervals of equal length.

Return type float

Examples

```
>>> simps(lambda x : 3*x**2,0,1,10)
1.0
```

Stolen from: https://www.math.ubc.ca/~pwalls/math-python/integration/simpsons-rule/

jax_cosmo.scipy.interpolate module

jax_cosmo.scipy.interpolate.**interp**(*x*, *xp*, *fp*)

Vectorized version of interp. Takes similar arguments as interp but with additional array axes over which interp is mapped.

Original documentation:

Simple equivalent of np.interp that compute a linear interpolation.

We are not doing any checks, so make sure your query points are lying inside the array.

TODO: Implement proper interpolation!

x, xp, fp need to be 1d arrays

jax_cosmo.scipy.ode module

jax_cosmo.scipy.ode.**odeint**(*fn*, *y0*, *t*)

My dead-simple rk4 ODE solver. with no custom gradients

CHAPTER 2

Indices and tables

- genindex
- modindex
- search

Python Module Index

j

Index