probfindiff

Nicholas Krämer

May 08, 2023

GETTING STARTED

Probabilistic numerical finite differences. Like traditional finite difference formulas, but with modelling flexibility and uncertainty quantification.

CHAPTER

ONE

INSTALLATION

Install via pip:

```
pip install probfindiff
```

or get the latest version from source:

```
pip install git+https://github.com/pnkraemer/probfindiff.git
```

Please be aware that during the early stages of development, some interfaces may be subject to change.

CHAPTER

TWO

A QUICK EXAMPLE

Finite difference methods estimate derivatives of functions from point-evaluations of said function. The same is true for probabilistic finite differences.

This set of notes explains the very basics of computing numerical derivatives with `probfindiff`. As a side quest, some basic design choices are explained.

[1]:
```
import jax.numpy as jnp

from probfindiff import central, differentiate, forward
```

2.1 First-order derivatives

At the heart of `probfindiff`, there is the function `differentiate()`, and a set of finite difference schemes. For example, to differentiate a function with a central scheme, compute the following.

[2]:
```
scheme, xs = central(dx=0.01)
dfx, _ = differentiate(jnp.sin(xs), scheme=scheme)

print(dfx, jnp.cos(0.0))
```

```
WARNING:absl:No GPU/TPU found, falling back to CPU. (Set TF_CPP_MIN_LOG_LEVEL=0 and␣
↪rerun for more info.)
```

```
1.0003637 1.0
```

The function `differentiate` acts on point-evaluations of a function on some grid-points. These points can be chosen by a user, but more often than not, they are coupled tightly to the scheme itself.

[3]:
```
print("xs =", xs)
print()
print("scheme =", scheme)
```

```
xs = [-0.01  0.    0.01]

scheme = FiniteDifferenceScheme(weights=DeviceArray([-5.0015533e+01, -6.9692903e-03,  5.
↪0022503e+01], dtype=float32), covs_marginal=DeviceArray(-0.00038028, dtype=float32),␣
↪order_derivative=DeviceArray(1, dtype=int32, weak_type=True))
```

The function `differentiate()` is self is so simple and lightweight, you could in fact implement it yourself.

[4]:
```
dfx = jnp.sin(xs) @ scheme.weights
print(dfx, jnp.cos(0.0))
```

```
1.0003637 1.0
```

The finite difference scheme expects that the array consists of function evaluations at a specific grid. This is important, because, for instance, smaller step-sizes imply different weights/coefficients, and different accuracy.

The requirement of acting only on discretised functions is different to many existing finite difference implementations, which behave more like automatic differentiation (i.e., they act on the function *as a function* and evaluate it internally).

Why? This design choice is deliberate. In many applications, e.g. differential equations, the number of function evaluations counts. Depending on the implementation, some functions can also be batched efficiently, while others cannot. To make this transparent, `probfindiff` lets a user evaluate their functions themselves. It is therefore closer to `np.gradient` than to automatic differentiation. (There are also some other advantages regarding types, compilation, and vectorisation, but this is left for a different tutorial.)

2.2 Higher-order derivatives

It is easy to compute higher-order derivatives by changing the scheme accordingly.

[5]:
```
scheme, xs = central(dx=0.01, order_derivative=2)
d2fx, _ = differentiate(jnp.sin(xs), scheme=scheme)

print(d2fx, -jnp.sin(0.0))
```

```
-1.1569691e-06 -0.0
```

2.3 Higher-order methods

To increase the accuracy of the approximation, the method-order can be increased freely.

[6]:
```
scheme, xs = central(dx=0.02, order_method=4)
dfx, _ = differentiate(jnp.sin(xs), scheme=scheme)

print(dfx, jnp.cos(0.0))
```

```
0.9998326 1.0
```

2.4 Forward, central, and backward schemes

While central schemes tend to be more accurate than forward and backward schemes, all three are available. For example, we can replace the central scheme with a forward scheme

[7]:
```
scheme, xs = forward(dx=0.02)
dfx, _ = differentiate(jnp.sin(xs), scheme=scheme)

print(dfx, jnp.cos(0.0))
```

```
1.0013572 1.0
```

2.5 What has been left out?

In all the examples above, we have ignored the second output of `differentiate()`: the uncertainty associated with the estimate. Its meaning, and how to make the most of it, are subject for a different tutorial.

CHAPTER

THREE

FINITE DIFFERENCES ON CUSTOM GRIDS

This tutorial explains how to compute finite difference approximations on custom grids.

[1]:
```
import jax.numpy as jnp

from probfindiff import backward, differentiate, from_grid
```

Recall how the usual output of finite difference schemes, for instance those resulting from `backward`, are a scheme and a grid. Subsequently, when applying the scheme, `probfindiff` assumes that the function has been evalauted at the grid.

[2]:
```
scheme, xs = backward(dx=0.1)
print(scheme)
print(xs)
```

```
WARNING:absl:No GPU/TPU found, falling back to CPU. (Set TF_CPP_MIN_LOG_LEVEL=0 and␣
↪rerun for more info.)
```

```
FiniteDifferenceScheme(weights=DeviceArray([ 15.01597  , -20.031385 ,   5.0154157],␣
↪dtype=float32), covs_marginal=DeviceArray(-5.543232e-05, dtype=float32), order_
↪derivative=DeviceArray(1, dtype=int32, weak_type=True))
[ 0.  -0.1 -0.2]
```

3.1 Custom schemes

Sometimes, we have a grid and want to compute a corresponding finite difference scheme. For example, when dealing with irregular geometries (circles, curves), or when specific function evaluations are readily available, and more evaluations are costly.

Luckily, there are ways to compute finite difference schemes from a grid.

[3]:
```
xs = jnp.array([-0.01, 0.0, 2.0])
scheme = from_grid(xs=xs)
print(scheme)
```

```
FiniteDifferenceScheme(weights=DeviceArray([-9.9635269e+01,  9.9632912e+01,  2.3571502e-
↪03], dtype=float32), covs_marginal=DeviceArray(-0.00106692, dtype=float32), order_
↪derivative=DeviceArray(1, dtype=int32, weak_type=True))
```

3.2 Where is x?

For the `from_grid()`, as well as its cousins `forward()`, `backward()`, etc., it is always assumed that the function's derivative shall be computed at the origin `x=0`. For instance, the grid `(-0.1, 0., 2.)` computes something like an unevenly-spaced central difference quotient, because the resulting differentiation scheme will approximate `f'(0)`.

[4]:
```
dfx, _ = differentiate(jnp.cos(xs), scheme=scheme)
print(dfx, -jnp.sin(0.0))
```

```
0.0016435911 -0.0
```

If you require the finite difference quotient at `x=x_0` instead, you can shift the evaluation points accordingly.

[5]:
```
dfx, _ = differentiate(jnp.cos(xs + 0.75), scheme=scheme)
print(dfx, -jnp.sin(0.75))
```

```
-0.6793945 -0.6816388
```

[ ]:

[ ]:

CHAPTER

FOUR

FINITE DIFFERENCES AND MODELLING

This notebooks explains a few of the subtleties of common assumptions behind finite difference schemes.

It will also outline one of the key advantages of `probfindiff` over competing packages: *making modelling explicit*.

[2]:
```
import jax.numpy as jnp

from probfindiff import central
```

Whenever you use `probfindiff`, remember that you are essentially building a Gaussian process model. The computation of the PN finite difference schemes assumes that the to-be-differentiated function f is

$$f \sim \mathrm{GP}(0, k)$$

for some covariance kernel function k. (This assumptions is implicit in non-probabilistic schemes – more on this later). This assumption is inherent in the `probfindiff` code. Central, forward, backward, and custom schemes automatically tailor to Gaussian covariance kernel functions.

[3]:
```
k_exp_quad = lambda x, y: jnp.exp(-jnp.dot(x - y, x - y) / 2.0)

scheme, xs = central(dx=1.0, kernel=k_exp_quad)
print(scheme)
```

```
WARNING:absl:No GPU/TPU found, falling back to CPU. (Set TF_CPP_MIN_LOG_LEVEL=0 and␣
↪rerun for more info.)
```

```
FiniteDifferenceScheme(weights=DeviceArray([-7.014634e-01,  4.714658e-08,  7.014634e-01],
↪ dtype=float32), covs_marginal=DeviceArray(0.14908189, dtype=float32), order_
↪derivative=DeviceArray(1, dtype=int32, weak_type=True))
```

Did you know that traditional finite difference coefficients $c = (1, -2, 1)$ implicitly assume that the function to-be-differentiated is a polynomial?

[4]:
```
k_poly = lambda x, y: jnp.polyval(x=jnp.dot(x, y), p=jnp.ones((3,)))
scheme, xs = central(dx=1.0, order_derivative=2, kernel=k_poly)
print(scheme.weights, jnp.allclose(scheme.weights, jnp.array([1.0, -2.0, 1.0])))
```

```
[ 1. -2.  1.] True
```

Whether this is right or wrong for your application, has to be decided by yourself. So next time you choose a finite difference scheme, please remember that you do not have to live like this, and that you can indeed compute finite difference formulas that are perfect for your model (and not build a model that uses some magic finite difference scheme).

CHAPTER

FIVE

PARTIAL DERIVATIVES AND OTHER AXIS MAGIC

Can `probfindiff` also do partial derivatives? Yes, it can do this and more!

[1]:
```
import jax.numpy as jnp

from probfindiff import central, differentiate_along_axis
```

5.1 Partial derivatives

Consider a function $f = f(x, y)$. To compute its partial derivative $\partial/\partial x f$, we can use finite differences. To this end, we build a meshgrid-style evaluation of f at the finite difference nodes (just as if we made a contour plot with matplotlib) and differentiate the resulting `(n,n)` array numerically.

[2]:
```
scheme, xs = central(dx=0.05)

fx = jnp.sin(xs)[:, None] * jnp.cos(jnp.zeros(1))[None, :]
dfdx_approx, _ = differentiate_along_axis(fx, axis=0, scheme=scheme)
print(dfdx_approx, jnp.cos(0.0) * jnp.cos(jnp.zeros(1)))
```

```
WARNING:absl:No GPU/TPU found, falling back to CPU. (Set TF_CPP_MIN_LOG_LEVEL=0 and␣
↪rerun for more info.)
```

```
[0.9995756] [1.]
```

5.2 Mixed derivatives

It is just as easy to compute mixed derivatives. For example, to compute

$$\frac{\partial^3}{\partial x \partial y^2} f(x, y)$$

we chain finite difference schemes together.

[3]:
```
scheme_x, xs = central(dx=0.1, order_derivative=1)
scheme_y, ys = central(dx=0.05, order_derivative=2)

fx = jnp.sin(xs)[:, None] * jnp.cos(ys)[None, :]
dfdx_approx, _ = differentiate_along_axis(fx, axis=0, scheme=scheme_x)
dfx_approx, _ = differentiate_along_axis(dfdx_approx, axis=0, scheme=scheme_y)
print(dfx_approx, -jnp.cos(0.0) ** 2)
```

```
-1.0064235 -1.0
```

If you've read the modelling tutorial, you will notice how this chain of applications implies a specific model. More specifically, the above is a good idea if the function f splits into the product

$$f(x, y) = f_1(x) f_2(y).$$

If not, there are better approaches. This will be left for a different tutorial.

5.3 Batched derivatives

Once we have the scheme, we can also use the `differentiate_along_axis()` function to compute batched finite difference evaluations. Since we set up the schemes independently of applying them, we can pick a scheme and apply it to a batch of function evaluations easily.

[4]:
```
scheme, xs = central(dx=0.01)
fx_batch = jnp.sin(xs)[:, None] * jnp.linspace(0.0, 1.0, 100)[None, :]
dfx_batch, _ = differentiate_along_axis(fx_batch, axis=0, scheme=scheme)

difference = dfx_batch - jnp.cos(0.0) * jnp.linspace(0.0, 1.0, 100)
print(difference.shape, jnp.linalg.norm(difference) / jnp.sqrt(difference.size))
```

```
(100,) 0.00021049284
```

CHAPTER

SIX

OBSERVATION NOISE

What do you do if your function evaluations are noisy? Plain finite difference schemes struggle in this setup.

[1]:
```
import jax.numpy as jnp
from jax import random

from probfindiff import central, differentiate

key = random.PRNGKey(seed=1)
```

```
WARNING:absl:No GPU/TPU found, falling back to CPU. (Set TF_CPP_MIN_LOG_LEVEL=0 and␣
↪rerun for more info.)
```

Noisy observations (even with very little noise) tend to mess up finite difference approximations.

[2]:
```
scheme, xs = central(dx=0.01)
noise = 1e-2 * random.normal(key, shape=xs.shape)

df, _ = differentiate(jnp.sin(xs) + noise, scheme=scheme)
print(df, jnp.cos(0.0))
```

```
1.5257977 1.0
```

Once we know which kind of error we are dealing with (noisy function evaluations), we can tune the FD scheme accordingly.

[3]:
```
scheme, xs = central(dx=0.01, noise_variance=1e-4)
df, _ = differentiate(jnp.sin(xs) + noise, scheme=scheme)
print(df, jnp.cos(0.0))
```

```
1.0176924 1.0
```

Mathematically, this is due to the fact that PN finite differences do some clever Gaussian process regression, which deals well with noise. The takeaway is: don't overfit your finite difference scheme. Adapt it to incorporate noise.

CHAPTER

SEVEN

BATCHED DIFFERENTIATION

Rarely does one have to compute only derivatives at single grid-points. More often than not, we want derivatives on a whole grid.

[1]:
```
import jax.numpy as jnp

from probfindiff import central, differentiate_along_axis, from_grid
```

7.1 Uniform grids

Let's say we need to compute the derivatives of a function f at a whole selection of grid-points. We can do this by exploiting the mechanisms of partial derivatives as follows. The only important thing to remember is that the schemes provided by `probfindiff` assume that the desired derivative is evaluated at zero, so we need to shift the finite difference grid appropriately.

[2]:
```
scheme, xs = central(dx=0.01)

grid = jnp.linspace(0.0, 1.0, num=12)

# Nonzero differentiation points
grid_fd = grid[:, None] + xs[None, :]

dfxs, _ = differentiate_along_axis(jnp.sin(grid_fd), axis=1, scheme=scheme)
print(dfxs, jnp.cos(grid))
```

```
WARNING:absl:No GPU/TPU found, falling back to CPU. (Set TF_CPP_MIN_LOG_LEVEL=0 and␣
↪rerun for more info.)
```

```
[1.0003637  0.9962324  0.9838748  0.9633885  0.9349485  0.8987875
 0.8552009  0.80455595 0.7472604  0.6837938  0.6146878  0.54050064] [1.         0.
↪9958706  0.9835166  0.9630399  0.93460965 0.8984607
 0.85489154 0.8042621  0.7469903  0.6835494  0.6144632  0.5403023 ]
```

7.2 Non-uniform grids

It is not surprising that the above procedure works equally well with non-uniform grids.

[3]:
```
xs = jnp.array([0.0, 0.3, 0.4])
scheme = from_grid(xs=xs)

grid = jnp.linspace(0.0, 1.0, num=12)
grid_fd = grid[:, None] + xs[None, :]

dfxs, _ = differentiate_along_axis(jnp.sin(grid_fd), axis=1, scheme=scheme)
print(dfxs, jnp.cos(grid))
```

```
[1.0196     1.0150714  1.0021594  0.98097146 0.9516821  0.9145313
 0.86982924 0.81794256 0.75930154 0.69438916 0.6237417  0.5479433 ] [1.         0.
↪9958706  0.9835166  0.9630399  0.93460965 0.8984607
 0.85489154 0.8042621  0.7469903  0.6835494  0.6144632  0.5403023 ]
```

7.3 Without redundant function evaluations

The downside of the above approach is that the function f has to be evaluated at a few redundant points. Chances are that the point evaluates are already part of the vector.

Can we be more efficient in this case? Yes, we can! While there is still much room for improvement (in terms of API and efficiency), the basics are accessible through convolutions.

[4]:
```
dx = 0.1
xs = jnp.arange(0.0, 2.0, step=dx)
scheme, _ = central(dx=dx)

# jax.numpy.convolve flips the second coefficient set.
# mode="valid" discards the meaningless points on the boundary
dfx_approx = jnp.convolve(jnp.sin(xs), jnp.flip(scheme.weights), mode="valid")
dfx_true = jnp.cos(xs)

print(dfx_approx)
print(dfx_true[1:-1])  # eliminate values we cannot compute in the above way
```

```
[ 0.99335146  0.9784374   0.95374703  0.9195273   0.87611985  0.8239579
  0.7635641   0.6955409   0.62056756  0.5393946   0.45283175  0.36174345
  0.26704264  0.16967201  0.07060671 -0.02916431 -0.12864399 -0.22683764]
[ 0.9950042   0.9800666   0.9553365   0.921061    0.87758255  0.8253356
  0.7648422   0.6967067   0.6216099   0.5403023   0.4535961   0.3623577
  0.26749876  0.16996716  0.0707372  -0.02919955 -0.12884454 -0.22720216]
```

Since central coefficients are not well-defined on the boundary of the grid, we only obtain the derivatives on the interior. For those, we could use forward/backward differences, or apply boundary conditions. Which one the correct solution is, depends on the application and is left for a different tutorial.

CHAPTER

EIGHT

MULTIVARIATE DERIVATIVES

Other notebooks have explained how to compute partial derivatives, but what if we want full gradients?

[1]:
```
import jax
import jax.numpy as jnp

from probfindiff import central, differentiate, from_grid, stencil
```

Let's define a function $f : R^d \to R$.

[2]:
```
f = lambda z: jnp.dot(z, z)
d = 4

x = jnp.arange(1.0, 1.0 + d)
df = jax.jacfwd(f)
```

```
WARNING:absl:No GPU/TPU found, falling back to CPU. (Set TF_CPP_MIN_LOG_LEVEL=0 and␣
↪rerun for more info.)
```

We have to extend the 1d scheme to a multivariate scheme. A multivariate scheme comes with a new set of coefficients and a new grid, that is adapted to the shape of the function.

[3]:
```
scheme, xs_1d = central(dx=0.01)
xs = stencil.multivariate(xs_1d=xs_1d, shape_input=(d,))
print(xs.shape)
```

```
(4, 4, 3)
```

The shape of `xs` is deliberate. The "final" dimension of the grid must correspond to the finite-difference-weight-multiplication. The shapes in the very front must correspond to the input and output shape of the domain of the function, because we aim to match the shapes of Jax' automatic differentiation behaviour. Therefore, the "evaluation domain" (which is the axis that will be "contracted" to `shape=()` by the scalar-valued function) must be `axis=-2` (the only axis left).

Let us evaluate the gradient numerically now.

[4]:
```
# Firstly, batch over the FD coefficients.
# Secondly, over the input shape(s).
f_batched = jax.vmap(jax.vmap(f, in_axes=-1), in_axes=0)
fx = f_batched(x[None, :, None] + xs)
dfx, _ = differentiate(fx, scheme=scheme)
print(dfx, df(x))
```

```
[2.0007117 4.0015507 6.002268  8.003107 ] [2. 4. 6. 8.]
```

The same can be done for *any* one-dimensional scheme.

[5]:
```
xs_1d = jnp.array([-0.1, -0.01, 0.0, 0.01, 0.1])
scheme = from_grid(xs=xs_1d)
xs = stencil.multivariate(xs_1d=xs_1d, shape_input=(d,))

fx = f_batched(x[None, :, None] + xs)
dfx, _ = differentiate(fx, scheme=scheme)
print(xs.shape, dfx, df(x))
```

```
(4, 4, 5) [1.9995394 3.9993134 5.9990873 7.9988513] [2. 4. 6. 8.]
```

The parameter `shape_input` already suggests that this mechanism extends to more complex schemes, such as Jacobians of truly multivariate functions. But this is content for a different tutorial.

CHAPTER

NINE

CALIBRATION AND MODEL SELECTION

Probabilistic numerical finite differences use the formalism of Gaussian process regression to derive the schemes. This brings with it the advantage of uncertainty quantification, but also the burden of choosing a useful prior model.

In this notebook, we will discuss the very basics of model selection and uncertainty quantification.

[1]:
```
import functools

import jax
import jax.numpy as jnp
import jax.scipy.stats

import probfindiff
from probfindiff.utils import kernel, kernel_zoo
```

First, a baseline. With a bad scale-parameter-estimate,, the error and uncertainty quantification are off.

[2]:
```
def k(*, input_scale):
    """Fix the input scale of an exponentiated quadratic kernel."""
    return functools.partial(
        kernel_zoo.exponentiated_quadratic, input_scale=input_scale
    )


dx = 0.1

# an incorrect scale messes up the result
scale = 100.0
scheme, xs = probfindiff.central(dx=dx, kernel=k(input_scale=scale))

f = lambda x: jnp.cos((x - 1.0) ** 2)
fx = f(xs)
dfx, variance = probfindiff.differentiate(fx, scheme=scheme)

dfx_true = jax.grad(f)(0.0)
error, std = jnp.abs(dfx - dfx_true), jnp.sqrt(variance)
print("Scale:\n\t", scale)
print("Error:\n\t", error)
print("Standard deviation:\n\t", std)
```

```
WARNING:absl:No GPU/TPU found, falling back to CPU. (Set TF_CPP_MIN_LOG_LEVEL=0 and␣
↪rerun for more info.)
```

```
Scale:
         100.0
Error:
         0.67733574
Standard deviation:
         3.8611143
```

We can tune the prior kernel to alleviate this issue. For example, we can compute the maximum-likelihood estimate of the input-scale θ. The goal is to find

$$\arg\max_{\theta} p(f_\theta(x_n) = f_n,\ n = 0, ..., N \mid \theta)$$

where f_θ is the prior Gaussian process, f_n are the observations of the to-be-differentiated function, and x_n are the finite difference grid points.

The problem is small, so let us be lazy and compute the minimum with a grid-search over a logarithmic space.

[3]:
```
@functools.partial(jax.jit, static_argnames=("kernel_from_scale",))
def mle_input_scale(*, xs_data, fx_data, kernel_from_scale, input_scale_trials):
    """Compute the maximum-likelihood-estimate for the input scale."""

    # Fix all non-varying parameters, vectorise, and JIT.
    scale_to_logpdf = functools.partial(
        input_scale_to_logpdf,
        fx_data=fx_data,
        kernel_from_scale=kernel_from_scale,
        xs_data=xs_data,
    )
    scale_to_logpdf_optimised = jax.jit(jax.vmap(scale_to_logpdf))

    # Compute all logpdf values for some trial inputs.
    logpdf_values = scale_to_logpdf_optimised(input_scale=input_scale_trials)

    # Truly terrible input scales lead to NaN values.
    # They are obviously not good candidates for the optimum.
    logpdf_values_filtered = jnp.nan_to_num(logpdf_values, -jnp.inf)

    # Select the optimum
    index_max = jnp.argmax(logpdf_values_filtered)
    return input_scale_trials[index_max]


@functools.partial(jax.jit, static_argnames=("kernel_from_scale",))
def input_scale_to_logpdf(*, input_scale, xs_data, fx_data, kernel_from_scale):
    """Compute the logpdf of some data given an input-scale."""

    # Select a kernel with the correct input-scale
    k_scale = kernel_from_scale(input_scale=input_scale)
    k_batch = kernel.batch_gram(k_scale)[0]

    # Compute the Gram matrix and evaluate the logpdf
    K = k_batch(xs_data, xs_data.T)
    return jax.scipy.stats.multivariate_normal.logpdf(
```

(continues on next page)

(continued from previous page)

```
        fx_data, mean=jnp.zeros_like(fx_data), cov=K
    )
```

[4]:
```
scale = mle_input_scale(
    xs_data=xs,
    fx_data=fx,
    kernel_from_scale=k,
    input_scale_trials=jnp.logspace(-3, 4, num=1_000, endpoint=True),
)
print("The optimised input scale is:\n\ts =", scale)
```

```
The optimised input scale is:
        s = 2.3462286
```

The resulting parameter estimate improves the calibration significantly.

[5]:
```
scheme, xs = probfindiff.central(dx=dx, kernel=k(input_scale=scale))
dfx, variance = probfindiff.differentiate(f(xs), scheme=scheme)

error, std = jnp.abs(dfx - dfx_true), jnp.sqrt(variance)
print("Scale:\n\t", scale)
print("Error:\n\t", error)
print("Standard deviation:\n\t", std)
```

```
Scale:
         2.3462286
Error:
         0.019150138
Standard deviation:
         0.0146484375
```

CHAPTER

TEN

API: PROBFINDIFF PACKAGE

10.1 Submodules

10.1.1 probfindiff.stencil

10.1.2 probfindiff.collocation

10.2 Subpackages

10.2.1 probfindiff.utils

Submodules

probfindiff.utils.autodiff

probfindiff.utils.kernel

probfindiff.utils.kernel_zoo

CHAPTER

ELEVEN

CONTRIBUTION GUIDE

Install `probfindiff` with all ci-related dependencies via

```
pip install .[ci]
```

Run all checks via

```
tox
```

or only run the tests via

```
tox -e pytest
```

or use tox (which also runs the linter, and the python-code-snippets in this readme).

```
tox
```

The CI checks for compliance of the code with black and isort, and runs the tests and the notebooks. To automatically satisfy the former, there is a pre-commit that can be used (do this once):

```
pip install pre-commit
pre-commit install
```

From then on, your code will be checked for isort and black compatibility automatically.

CHAPTER

TWELVE

RUNNING THE EXAMPLE NOTEBOOKS

To run the example notebooks, additional dependencies need to be installed via

```
pip install .[examples]
```

For example: `jupyter`.

CHAPTER

THIRTEEN

INDICES AND TABLES

- genindex
- modindex
- search