NIFTy Introduction

Jakob Roth, Andreas Popp, Ananya Shankar

May 28, 2026

CONTENTS

Introduction

This book provides an introduction to the Bayesian inference library NIFTy. NIFTy is designed to enable inference in very high-dimensional posterior distributions, as required for image or 3D volume reconstruction applications. It has already been successfully applied to problems with hundreds of millions of parameters. To support such applications, NIFTy includes at its core scalable Gaussian process models and fast variational inference algorithms. There are two variants of NIFTy, called `NIFTy.re` and `NIFTy.cl`. This introduction focuses on the JAX-based variant `NIFTy.re`. However, many of the concepts introduced here also apply to `NIFTy.cl`.

The content of this book is based on the example notebooks available on the NIFTy webpage. In addition, the webpage includes further resources such as the API reference. You can also find additional introductory material in the demos folder of the GitLab repository. The source code of NIFTy and installation instructions are available on the GitLab page.

CHAPTER

ONE

NIFTY MODELS

This notebook provides an overview of the theory and implementation of statistical models in NIFTy. The first part introduces the mathematical background on how prior models are handled in NIFTy. The subsequent section discusses the implementation of prior models, and the final section focuses on likelihoods.

1.1 Introduction

Bayesian statistics allows us to combine the likelihood, which contains new information such as measurements, with prior knowledge encoded in a prior distribution. In NIFTy, the quantity we want to infer is typically called the signal s, and the data from which we want to infer s is denoted by d. Using this notation, Bayes' Theorem can be written as

$$P(s|d) = \frac{P(d|s)P(s)}{P(d)},$$

where $P(s|d)$ is the posterior, $P(d|s)$ the likelihood, and $P(s)$ the prior. The term $P(d)$, called the evidence, acts as a normalization constant for the posterior and is often ignored in NIFTy.

To reconstruct a signal s from a dataset d with NIFTy, you will need to code a prior model, a likelihood, and then run an inference algorithm. This notebook is about prior and likelihood models in NIFTy. Specifically, we will discuss the mathematical background of how priors are handled in NIFTy, as well as some implementation details for priors and likelihoods. In the next notebook, you will learn about how to infer the posterior distribution once you have a NIFTy model.

1.2 Prior Models in NIFTy

In many real-world applications, the prior distribution we want to impose on the quantity we infer (in NIFTy often called signal s) can be quite complicated. Nevertheless, the variational inference methods used in NIFTy to obtain the posterior distribution are built around Gaussian distributions. For this reason, a direct variational inference approximation of the posterior of the signal s would lead to a significant approximation error. To circumvent this problem, NIFTy requires the user to reparameterize the model such that the model parameters are a priori standard normal distributed. In the NIFTy language, the reparameterized model parameters are often called ξ, and the signal $s(\xi)$ becomes a function of the new standardized parameters. The space of the standardized parameters is called the latent space.

This notebook briefly introduces the mathematical foundation for standardizing models and explains how to code such models in NIFTy to map to the desired prior distribution.

1.2.1 Standardized Models

The inference algorithms of NIFTy assume that all parameters of the model are a priori standard normally distributed. We often denote these standard normally distributed parameters with ξ. If we now have a signal s that we want to reconstruct from some data d, but want to impose a non-standard normal prior, then we have to construct a function transforming from the standard normal distribution to the desired prior distribution.

Mathematically, such a mapping always exists and can be constructed from the Cumulative Density Function (CDF) of the Gaussian CDF_G and the inverse Cumulative Density Function $\text{CDF}^{-1}_{P(s)}$ of the desired distribution $P(s)$. It can be shown that if ξ is standard normally distributed $P(\xi) = G(0, 1)$, then

$$s(\xi) = (\text{CDF}^{-1}_{P(s)} \circ \text{CDF}_G)(\xi)$$

is distributed according to $P(s)$. Such mappings between parameters with a "simple" distribution and parameters with a more complex distribution are widely used in the statistics and machine learning community. They are, for example, also known as the reparametrization trick and are the core idea of inverse transform sampling.

Bayes' Theorem expressing the posterior distribution $P(s|d)$ in terms of the likelihood $P(d|s)$ and the prior $P(s)$

$$P(s|d) = \frac{P(d|s)P(s)}{P(d)},$$

can now be rewritten in terms of ξ

$$P(\xi|d) = \frac{P(d|s(\xi))P(\xi)}{P(d)},$$

with $P(\xi)$ being the standard normal distribution. For a more detailed mathematical introduction tailored towards the application in NIFTy, see arXiv:1812.04403.

To summarize, all the complexity of the desired prior distribution on s is now encoded in the mapping $\xi \to s$. The posterior inference algorithms discussed in the next notebook will infer the posterior distribution $P(\xi|d)$. Via the mapping from $\xi \to s$, the posterior of ξ determines the posterior distribution for s.

1.2.2 Implementation in NIFTy

The previous section introduced the mathematical foundation of standardized prior models. In this section, we will focus on the implementation in NIFTy.

As discussed, NIFTy always assumes that the model is standardized such that $P(\xi)$ is standard normally distributed. For this reason, NIFTy will automatically construct the corresponding standard Gaussian prior $P(\xi)$ for the parameters of the likelihood $P(d|s(\xi))$ the user has implemented. What NIFTy cannot do automatically is construct the mapping $\xi \to s$ and the likelihood itself $P(d|s)$, as these depends on the prior the use wants to impose and the statistics of the data. In this section, we discuss with an example how to implement $\xi \to s$. Afterwards, we will explain the implementation of $P(d|s)$.

Example Model

Implementation details of NIFTy models are best explained with examples. In this notebook, we will discuss a simple example: reconstructing the slope and offset of a linear function from measured data points. In other words, Bayesian linear regression. This simple example is ideal to get started with the NIFTy syntax. You can find more advanced examples showcasing prior models for image and volume reconstructions in the demos folder.

For our example, let us assume that we have measured the data $\vec{d} \in \mathbb{R}^N$ at some locations $\vec{x} \in \mathbb{R}^N$, and that there is a relation between $\vec{d}$ and $\vec{x}$ following the form

$$\vec{d} = a\vec{x} + b\vec{1} + \vec{n},$$

with a and b being some unknown scalar parameters that we want to infer. Thus the parameters a, b are our signal that we want to reconstruct from the data. n stands for some additive noise in the measurement. Furthermore, $\vec{1}$ symbolizes a vector with all entries being equal to 1. Our goal is now to obtain the posterior distribution $P(a, b|d)$ of the parameters a and b. To do so, we will first need to code a standardized prior model and a likelihood function, which we will do in this notebook. The next notebook will demonstrate how to obtain the posterior given the NIFTy model.

To implement the likelihood function, we code a generative model mapping from some latent a priori standard Gaussian distributed parameters $\vec{\xi}$ to the parameters $a(\vec{\xi})$ and $b(\vec{\xi})$ and then mapping to the predicted values of $\vec{y} = a(\vec{\xi})\vec{x} + b(\vec{\xi})\vec{1}$. Thus, in the end, we will have a function mapping $\vec{\xi}$ to the corresponding values $\vec{y}$.

Before starting with the actual implementation, let us import NIFTy and the relevant JAX libraries. Furthermore, we activate the float64 precision in JAX, which is recommended for NIFTy.

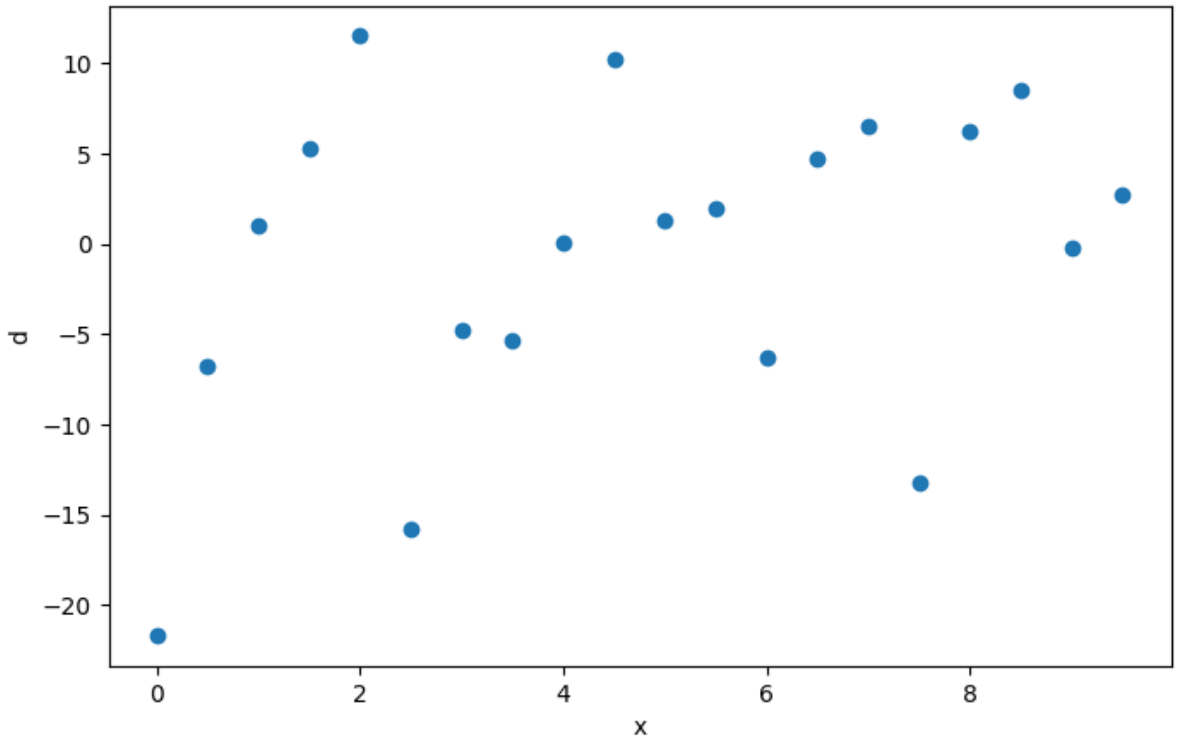
```
import nifty.re as jft

import numpy as np

import jax
import jax.random as random

%matplotlib inline
import matplotlib.pyplot as plt

plt.rcParams["figure.dpi"] = 100

jax.config.update("jax_enable_x64", True)
```

Furthermore, let us define the positions x at which we have measured and the actual data d we obtained. In an application to real data, you would load your dataset here.

```
x, d = np.loadtxt("data.txt", delimiter="\t", skiprows=1, unpack=True)
```

A first good step when dealing with new data is to visualize it. To visualize our data, we will plot the locations at which we have measured on the x-axis and the corresponding datapoint on the y-axis.

```
plt.figure(figsize=(8, 5))
plt.scatter(x, d)
plt.xlabel("x")
plt.ylabel("d")
plt.show()
```



For this data, we now want to do a Bayesian linear regression. Let us assume that we have some prior knowledge of the value of the parameters a and b. Namely, let us assume about a that we already know that it must be a positive number and that we believe that its value is around 4 with an uncertainty of ± 3. A suitable probability distribution for encoding this prior knowledge is the Log-Normal distribution. NIFTy already implements models for commonly used probability distributions, such as the Log-Normal distribution. A list of all implemented probability distributions can be found on the API reference page.

We specify a NIFTy model for a Log-Normal distribution with `mean=4` and `std=3`. The meaning of the parameter `name` will be explained in the cell below.

```
a = jft.LogNormalPrior(mean=4, std=3, name="a_input")
```

`a` is now a NIFTy model implementing the mapping of a standard normal distributed ξ to the Log-Normal distributed parameters a. Besides this mapping, the NIFTy model implements additional useful functionality, including information about the input and output of the mapping function. Specifically, `a.domain` contains information on how the input to the mapping function should be formatted.

```
print(a.domain)
```

```
{'a_input': ShapeWithDtype(shape=(), dtype=<class 'numpy.float64'>)}
```

As we see, the input to the model `a` should be a Python dictionary. This dictionary should include a key named `a_input` containing a single float. This float will be mapped to the log-normal distributed quantity. Why it is useful that the input is wrapped inside a dictionary will become clear in the discussion below. The fact that the key containing the input for the model `a` is named `a_input` is because we set this for the `name` in the initialization of the model `a`. We will also see in the later discussion that different components of our prior model, such as the models a and b, must use different keys.

The `target` property contains information about the model's output. We see that the output of the model is a single float:

```
print(a.target)
```

```
ShapeDtypeStruct(shape=(), dtype=float64)
```

We can also create some test input and apply the model `a` to it to showcase how to apply the model:

```
a_test_input = {"a_input": 0.0}
res = a(a_test_input)
print(res)
```

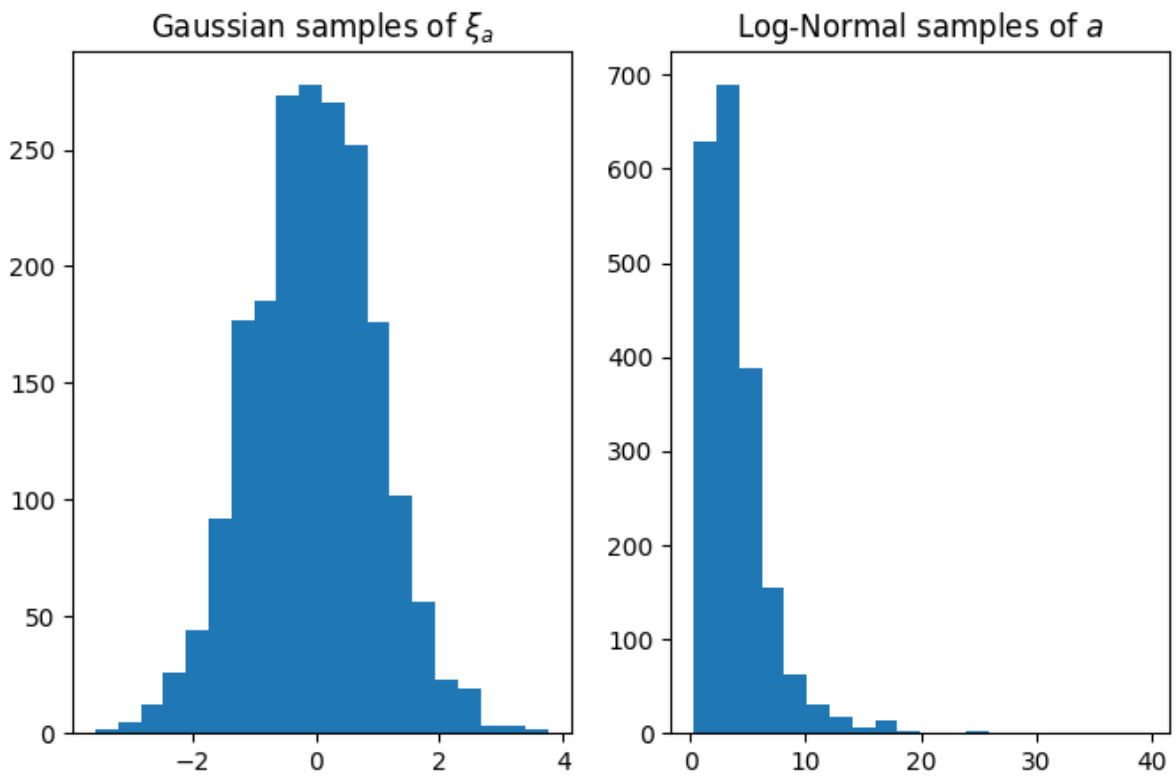
```
3.2
```

To showcase how the Gaussian distribution on the domain of `a` is mapped to a Log-Normal distribution, let us draw Gaussian samples and pass them into `a`. Before doing so, we first need to introduce the JAX random number generator.

To draw random numbers in NIFTy, we use the JAX random number generator. You might want to learn more about the JAX random number generator on the JAX webpage, as the JAX random number generator works slightly differently than the NumPy random number generator. In essence, to get a random number, you first generate a random key. Given this key, deterministic random numbers are created. To generate new random numbers, the key can be split into a new key and a subkey.

Below is the code to generate random Gaussian samples for ξ_a and compute the corresponding samples for $a(\xi_a)$. To do so, we first set the random seed and then generate the initial random key for JAX. Afterwards, we generate 2000 random samples in a for loop. Inside the loop, we first split the JAX random key to get new random numbers in each iteration. Then we use the split random key to get a Gaussian random sample on the domain of `a` with `jft.random_like(subkey, a.domain)`. The Gaussian latent sample is then transformed into a sample of `a` by computing `a(latent_sample)`. Finally, we append both the latent sample and the corresponding sample from `a` to lists. After the for loop, we plot the lists of samples as histograms with matplotlib. The left plot visualizes the latent Gaussian samples, while the right plot shows the log-normal distributed samples from `a`.

```
seed = 42
key = random.PRNGKey(seed)

latent_sample_list = []
a_sample_list = []
for i in range(2000):
    key, subkey = random.split(key)
    latent_sample = jft.random_like(subkey, a.domain)
    a_sample = a(latent_sample)
    latent_sample_list.append(latent_sample["a_input"])
    a_sample_list.append(a_sample)

fig, axs = plt.subplots(nrows=1, ncols=2, figsize=(8, 5))
axs[0].hist(latent_sample_list, bins=20)
axs[0].set_title(r"Gaussian samples of $\xi_a$")
axs[1].hist(a_sample_list, bins=20)
axs[1].set_title(r"Log-Normal samples of $a$")
plt.show()
```



Besides the prior model for a, we have to specify a model for b. Let us assume that we want to use a Gaussian prior with a mean of 0 and a standard deviation of 3 for b.

```
b = jft.NormalPrior(mean=0, std=3, name="b_input")
```

As for `a`, we can look at the domain and target properties of `b`, giving information on how the input to `b` should look, and what the output will be:

```
print(b.domain)
print(b.target)
```

```
{'b_input': ShapeWithDtype(shape=(), dtype=<class 'numpy.float64'>)}
ShapeDtypeStruct(shape=(), dtype=float64)
```

Analogous to `a`, the input should be a dictionary with the value of the `name` parameter as a key. The output of `b` will be a single float.

Now we have implemented the generative models for $\xi \to s$. Specifically, we have implemented the models $\xi_a \to a$ and $\xi_b \to b$. The next step is to code a model for linear regression. Specifically, what we have to code is a model that predicts $y(\xi_a, \xi_b) = a(\xi_a)x + b(\xi_b)$ for given input (ξ_a, ξ_b) and given locations x. While for the models `a` and `b` we could use already existing models in NIFTy, this time we have to code our own model.

In its most basic form, a NIFTy model can simply be a Python function that implements the mapping from standard normal distributed parameters to the desired quantity. In our example case, the final quantity our model should map to is $y(\xi_a, \xi_b)$. Nevertheless, very often it is helpful to not only implement the function itself but also store information about the input and output of the model. For the models `a` and `b`, we already looked at this information, namely the properties `.domain` for the input and `.target` for the output. To help users implement custom models that also contain information about the domain and target, NIFTy provides the base class `jft.Model`, from which custom models can inherit. Below is an implementation of our model. In the following, we will explain the implementation. However, if you are not familiar with object-oriented programming in Python, we recommend reading one of the numerous tutorials on the web first.

```
class LinearModel(jft.Model):
    def __init__(self, x, a, b):
        self.x = x
        self.a = a
        self.b = b
        super().__init__(domain=a.domain | b.domain, white_init=True)

    def __call__(self, inp):
        return self.x * self.a(inp) + self.b(inp)


my_model = LinearModel(x, a, b)
```

The class `LinearModel` inherits from the base class `jft.Model` and implements our model $\vec{y}(\xi_a, \xi_b) = a(\xi_a)\vec{x} + b(\xi_b)$ for the data. To the `__init__` function of the class, we pass the locations x at which we have measured, together with our prior models for the parameters a and b. `x`, `a`, and `b` are then stored as properties of the model. Furthermore, the `__init__` method sets the domain of the model by calling the `__init__` of the `jft.Model` class. What exactly the domain needs to be depends on the details of the `__call__` function. For this reason, we will first discuss the `__call__` function before we come back to the domain.

The `__call__` function implements the actual model. In our case, the `__call__` function implements the mapping from the latent parameters ξ_a and ξ_b to the predicted values for y. Expressed as a formula, the call function computes: $(\xi_a, \xi_b) \to \vec{y} = a(\xi_a)\vec{x} + b(\xi_b)\vec{1}$. In code, the `__call__` function receives (besides `self`) the variable `inp` as input, which contains the values of ξ_a and ξ_b. Then the variable `inp` is passed to the models `a` and `b`. Afterwards, the output of `a(inp)` is multiplied by `x`, and `b(inp)` is added. Both `a` and `b` expect their input to be a Python dict containing the keys `a_input` and `b_input`, respectively. Therefore, to make this code line work without an error, `inp` also needs to be a Python dict with at least the keys `a_input` and `b_input`. For both these keys, the `inp` dict should contain a single float with the values of ξ_a and ξ_b. This determines what the domain of the model should be, namely, the domain of the model is a Python dict with the keys `a_input` and `b_input`. Thus, the domain is simply the union of the domains of the models `a` and `b`. In the call to the `__init__` method of `jft.Model`, we additionally set `white_init=True`. This is optional, but instructs NIFTy to initialize the inference of our model with uncorrelated standard Gaussian random numbers.

```
print(a.domain | b.domain)
```

```
{'a_input': ShapeWithDtype(shape=(), dtype=<class 'numpy.float64'>), 'b_input':␣
↪ShapeWithDtype(shape=(), dtype=<class 'numpy.float64'>)}
```

This union `a.domain | b.domain` is then passed as the `domain` to the init method of `jft.Model`, from which our `LinearModel` has inherited from. The `jft.Model` class implements the functionality that our model now also has a property `domain`.

```
print(my_model.domain)
```

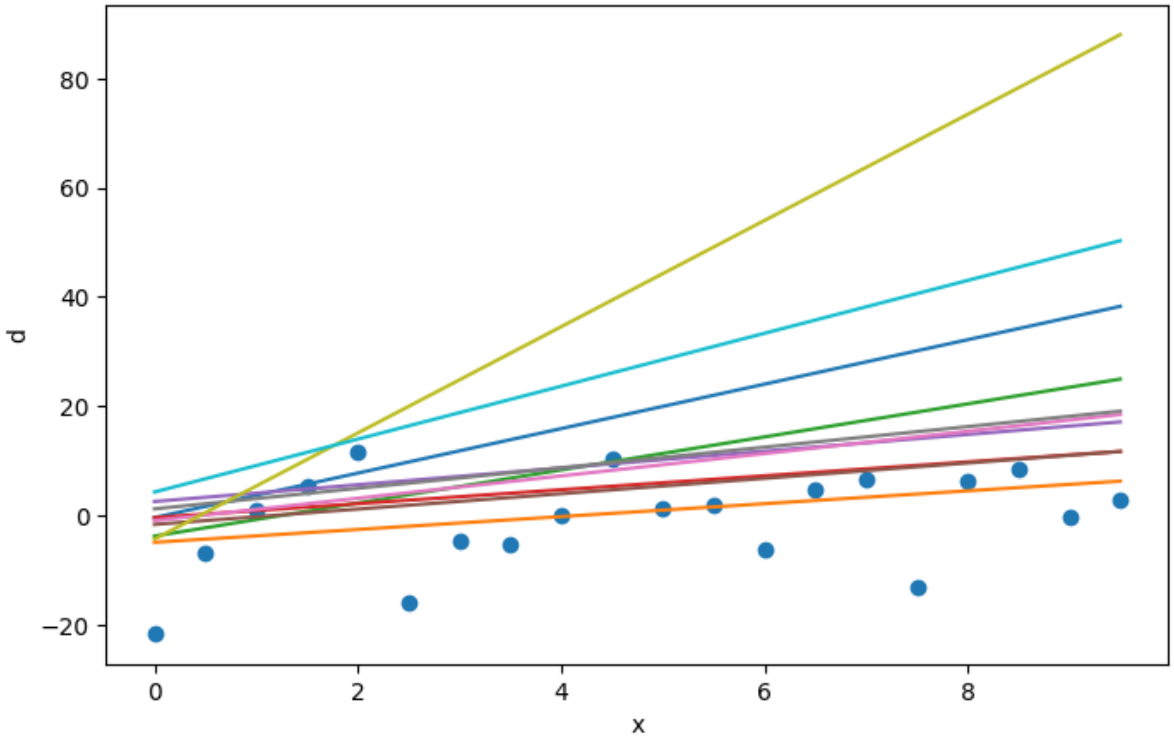
```
{'a_input': ShapeWithDtype(shape=(), dtype=<class 'numpy.float64'>), 'b_input':␣
↪ShapeWithDtype(shape=(), dtype=<class 'numpy.float64'>)}
```

That the domain of our model is also a Python dict is not a necessity, but a consequence of how we have implemented the call function. If we had implemented the call function differently, the domain could also have been something else. For example, if the call function were to evaluate the model like this: `self.x * self.a(inp[0]) + self.b(inp[1])`, then `inp` would need to be a Python list with the first element containing the input for `a` and the second element containing the input for `b`. Correspondingly, in this case, we should have set the `domain` to `[a.domain, b.domain]`. However, especially for large models, coding the call methods such that the input can be a dict is far more convenient than using a list or other Python data containers. For example, when using Python lists as the domain of all models, it is significantly more error-prone to add new components to a model, as one must ensure that each component

still receives its input from the correct index location. When using Python dictionaries for the input, this is much easier, as we just have to ensure that the new components use distinct keys in the dictionary.

Before continuing with implementing a likelihood, let us once more visualize our model by plotting prior samples from it. This time, we plot prior samples of our model alongside the data. We start with a scatter plot of our data. Then, we draw 10 prior samples of our model using a for loop and add them to the plot. Drawing a prior sample from the full model is very similar to drawing a prior sample from `a`, as discussed above. Again, we split the random key, generate a latent sample, and transform this latent sample into a sample of our model.

```
plt.figure(figsize=(8, 5))
plt.scatter(x, d)

for i in range(10):
    key, subkey = random.split(key)
    latent_sample = jft.random_like(subkey, my_model.domain)
    y_sample = my_model(latent_sample)
    plt.plot(x, y_sample)

plt.xlabel("x")
plt.ylabel("d")
plt.show()
```



1.3 Likelihood Models in NIFTy

To summarize, so far we have coded a model taking (ξ_a, ξ_b) as an input and computing the corresponding $y(\xi_a, \xi_b) = a(\xi_a)\vec{x} + b(\xi_b)\vec{1}$. As a next step, we will implement the actual likelihood function, allowing us to evaluate $P(d|\xi_a, \xi_b)$. For this, we need to assume something about the distribution of the noise in the measurement equation $d = y + n = a\vec{x} + b\vec{1} + n$. For this introductory example, we will assume that n is Gaussian distributed with a standard deviation of 10 and all measurements are uncorrelated. For the Gaussian case, we can use the `jft.Gaussian` likelihood. Likelihoods for other noise distributions and measurement setups can be found on the API reference page.

Assuming that n is Gaussian distributed, the likelihood is given by

$$P(d|\xi_a, \xi_b) = P(d|y(\xi_a, \xi_b)) = P(n = d - y(\xi_a, \xi_b)) = G(d - y, 10^2).$$

Thus, the likelihood is a Gaussian distribution with the mean being the data d and the covariance being the covariance of the noise, which is, in our case, a diagonal matrix with all values on the diagonal being $10^2 = 100$. For numerical reasons, NIFTy does not operate directly on the likelihood itself, but instead uses the negative logarithm of it. In NIFTy language, the negative logarithm is called the Hamiltonian and denoted by H. In formulas the likelihood Hamiltonian is given by

$$H(d|\xi_a, \xi_b) = -\ln(P(d|\xi_a, \xi_b)).$$

The corresponding NIFTy code looks like this:

```
cov = 10**2
noise_cov_inv = lambda x: x / cov  # diagonal matrix
lh = jft.Gaussian(data=d, noise_cov_inv=noise_cov_inv).amend(my_model)
```

```
assuming a diagonal covariance matrix;
setting `std_inv` to `cov_inv(ones_like(data))**0.5`
```

Conceptually, as well as in the implementation, the likelihood `lh` shares many similarities with normal models in NIFTy. For example, the model and the likelihood are a mapping of the same latent parameters (ξ_a, ξ_b). We can verify this by looking at the `.domain` properties of the likelihood and model:

```
print("domain of model: ", my_model.domain)
print("domain of lh: ", lh.domain)
```

```
domain of model:  {'a_input': ShapeWithDtype(shape=(), dtype=<class 'numpy.float64
 ↪'>), 'b_input': ShapeWithDtype(shape=(), dtype=<class 'numpy.float64'>)}
domain of lh:  {'a_input': ShapeWithDtype(shape=(), dtype=<class 'numpy.float64'>),
 ↪ 'b_input': ShapeWithDtype(shape=(), dtype=<class 'numpy.float64'>)}
```

The likelihood and the model both have a `.target` property indicating the type of the output. While the model maps to an array of 20 floats (the values of y), the likelihood maps to the log of the probability, which is a single float.

```
print("target of model: ", my_model.target)
print("target of lh: ", lh.target)
```

```
target of model:  ShapeDtypeStruct(shape=(20,), dtype=float64)
target of lh:  ShapeDtypeStruct(shape=(), dtype=float64)
```

1.4 Summary

This notebook introduces the concept of standardized generative models. These models encode a potentially complex probability distribution of some parameters via a non-linear mapping from independent and identically distributed (i.i.d.) Gaussian-distributed parameters $\vec{\xi}$ to the actual parameters of interest. NIFTy builds upon this concept. NIFTy assumes for all parameters of a model a standard Gaussian prior, and a non-Gaussian distribution can be encoded in the form of a generative model in the likelihood function. The sections above have introduced how to code such a likelihood, including a generative model. In the next notebook, we will discuss how to access the corresponding posterior distribution with NIFTy.

CHAPTER

TWO

INFERENCE WITH NIFTY

This notebook is the continuation of the previous NIFTy models *notebook*. While the previous notebook gives a basic introduction to prior models and likelihoods in NIFTy, this notebook showcases how to obtain the posterior distribution for a given model in NIFTy.

After a brief mathematical introduction to the inference techniques used in NIFTy, this notebook will proceed with the linear regression example from the *previous notebook*. For this reason, please read the introduction to generative models first.

2.1 Recap

The previous notebook introduced a NIFTy model for linear regression in one dimension. Specifically, the model was

$$\vec{d} = a\vec{x} + b\vec{1} + \vec{n},$$

with $\vec{d}$ being the measured data, $\vec{x}$ the locations at which we measured, $\vec{n}$ some unknown noise in the measurements, and a and b the scalar parameters of the linear function we fit. We imposed a lognormal prior on a and a normal prior on b, both in the form of a standardized model as required by NIFTy. This means we have a priori standard normally distributed latent parameters $\vec{\xi} = (\xi_a, \xi_b)$ which are mapped to $a(\xi_a), b(\xi_b)$ with a mapping constructed such that a and b have the desired prior distributions.

In this notebook, we will infer the posterior distribution of $\vec{\xi}$, which determines the posterior for $a(\xi_a)$ and $b(\xi_b)$.

2.1.1 Code for the NIFTy Model

Below you can find the NIFTy code for the linear regression model we developed in the last notebook. In this notebook, we will not explain the NIFTy code again. Thus, if you are not familiar with NIFTy and have not read the previous notebook, please do so first.

```
import nifty.re as jft

import numpy as np

import jax
import jax.numpy as jnp
import jax.random as random

%matplotlib inline
import matplotlib.pyplot as plt

plt.rcParams["figure.dpi"] = 100
```

(continues on next page)

(continued from previous page)

```
# enable float64 precision
jax.config.update("jax_enable_x64", True)


# initialize JAX random key
seed = 42
key = random.PRNGKey(seed)

# measurement locations and data
x, d = np.loadtxt("data.txt", delimiter="\t", skiprows=1, unpack=True)


# models for a and b
a = jft.LogNormalPrior(mean=4, std=3, name="a_input")
b = jft.NormalPrior(mean=0, std=3, name="b_input")


# linear regression model
class LinearModel(jft.Model):
    def __init__(self, x, a, b):
        self.x = x
        self.a = a
        self.b = b
        super().__init__(domain=a.domain | b.domain, white_init=True)

    def __call__(self, inp):
        return self.x * self.a(inp) + self.b(inp)


my_model = LinearModel(x, a, b)


# likelihood
cov = 10**2
noise_cov_inv = lambda x: x / cov  # diagonal matrix
lh = jft.Gaussian(data=d, noise_cov_inv=noise_cov_inv).amend(my_model)
```

```
assuming a diagonal covariance matrix;
setting `std_inv` to `cov_inv(ones_like(data))**0.5`
```

To summarize, the above code implements a generative model that maps the latent parameters (ξ_a, ξ_b) to $\vec{y} = a(\xi_a)\vec{x} + b(\xi_b)\vec{1}$, and defines the likelihood $P(d|\xi_a, \xi_b)$. The generative process for a and b is designed such that the latent parameters ξ_a and ξ_b follow standard normal (unit Gaussian) priors.

2.2 Theory

Using Bayes' Theorem, we can express the posterior distribution $P(\vec{\xi}\,|\,d)$ corresponding to the above prior and likelihood as:

$$P(\vec{\xi}\,|\,d) = \frac{P(d\,|\,\vec{\xi})\,P(\vec{\xi})}{P(d)},$$

where the evidence $P(d)$ is given by $P(d) = \int d\vec{\xi}\,P(d\,|\,\vec{\xi})\,P(\vec{\xi})$.

For NIFTy models, the prior probability $P(\vec{\xi})$ is always a standard Gaussian. Bayes' Theorem can also be formulated in terms of information Hamiltonians, which are defined as the negative logarithm of the corresponding probability: $H = -\ln P$. For numerical reasons, NIFTy always performs computations in terms of Hamiltonians. Expressed in Hamiltonians, Bayes' Theorem becomes:

$$H(\vec{\xi}\,|\,d) = H(d\,|\,\vec{\xi}) + H(\vec{\xi}) - H(d).$$

While Bayes' Theorem provides a mathematically exact expression for the posterior distribution, directly extracting information from it is often challenging.

The primary difficulty lies in numerical integration, which becomes computationally intractable in high-dimensional spaces. For example, evaluating the evidence $P(d) = \int d\vec{\xi}\,P(d\,|\,\vec{\xi})\,P(\vec{\xi})$ requires integration over $\vec{\xi}$. Similarly, computing summary statistics, such as the posterior mean $\langle a\rangle = \int d\vec{\xi}\,a(\vec{\xi})\,P(\vec{\xi}\,|\,d)$, also requires integration over $\vec{\xi}$.

In our example, $\vec{\xi} = (\xi_a, \xi_b)$ is only two-dimensional, making numerical integration feasible. However, in many real-world problems, the number of parameters is much larger, rendering direct numerical integration impractical. Therefore, Bayesian inference often relies on advanced algorithms to avoid explicit high-dimensional integration.

2.2.1 Maximum a posteriori

The simplest, though often inaccurate, inference method is to compute only the maximum of the posterior distribution and use this as an estimate. This approach, known as maximum a posteriori (MAP) estimation, is computationally cheap and easy to implement.

To maximize $P(\vec{\xi}\,|\,d)$ (or equivalently minimize $H(\vec{\xi}\,|\,d) = -\ln P(\vec{\xi}\,|\,d) = H(d\,|\,\vec{\xi}) + H(\vec{\xi}) - H(d)$), we do not need to compute the evidence $H(d)$, as it is constant with respect to $\vec{\xi}$. Therefore, no integration over the $\vec{\xi}$ space is required. If $H(\vec{\xi}\,|\,d)$ is differentiable with respect to $\vec{\xi}$, gradient-based minimizers can efficiently find the minimum. This makes MAP estimation a very fast method for accessing posterior information.

However, MAP estimation has significant drawbacks, which is why more accurate, though computationally expensive, inference methods are often preferred:

1. **No Uncertainty Quantification:** MAP yields only a point estimate and does not provide information about the uncertainty or spread of the posterior.
2. **Volume Effects in High Dimensions:** For high-dimensional distributions, the mode (maximum density point) is not necessarily representative of typical samples from the posterior. This is because MAP considers only the probability density, not the volume of regions in parameter space. A region with moderate density but large volume may contribute more to the overall posterior probability mass than the small region near the MAP point. This "volume effect" becomes increasingly dominant as the number of dimensions grows, because the total volume grows exponentially with the number of dimensions.

Due to these limitations, MAP estimation is often insufficient, and more advanced inference algorithms are required. Two prominent alternatives are:

- **Monte Carlo Sampling:** Generates samples from the posterior and uses them to compute statistics such as the posterior mean.

- **Variational Inference:** Approximates the posterior with a simpler distribution, optimizing the parameters of this approximation to best match the true posterior.

While Monte Carlo methods can achieve high accuracy if enough samples are drawn, variational inference is typically computationally more efficient. For this reason, NIFTy primarily relies on variational inference.

2.2.2 Variational Inference

This section explains the basic idea behind the variational inference algorithms of NIFTy. For a detailed mathematical introduction, please refer to the original publications, specifically arXiv:2105.10470 and, for an earlier version, arXiv:1901.11033.

The idea of variational inference is to approximate the posterior distribution with a simpler distribution. The variational inference algorithms in NIFTy are built around Gaussian distributions, approximating the posterior with a Gaussian. The variational parameter optimized during this process is the mean of the Gaussian. The covariance is not directly optimized but is instead constructed from the inverse Fisher information metric, which leverages curvature information of the posterior. Thereby the variational inference algorithms of NIFTy can capture posterior correlations.

In the Metric Gaussian Variational Inference (MGVI) algorithm (arXiv:1901.11033), this Gaussian approximation is performed directly in the space of $\vec{\xi}$. The Geometric Variational Inference (geoVI) method (arXiv:2105.10470) applies an additional nonlinear transformation of the $\vec{\xi}$ space to make the true posterior more Gaussian, thereby improving the accuracy of the approximation.

From information theory, a distance measure for probability distributions can be derived that quantifies their similarity. This measure, known as the Kullback-Leibler (KL) divergence, serves as the cost function in variational inference. In NIFTy, the KL divergence between the true posterior and the Gaussian approximation is minimized with respect to the posterior mean of the approximating Gaussian.

The closer the true posterior distribution is to a Gaussian, the more accurate the approximation becomes. This is one reason why NIFTy enforces a standard Gaussian prior on $\vec{\xi}$, while non-Gaussianities are captured in the mapping $\xi \to s$. The idea is that in a coordinate system where the prior is Gaussian, the posterior is also relatively close to a Gaussian distribution. In particular, for degrees of freedom that are unconstrained or only weakly constrained by the likelihood, the posterior remains Gaussian, and the variational inference algorithms do not introduce significant approximation errors.

In the following, we briefly outline the variational inference procedure used in NIFTy. For a more thorough introduction to variational inference, see the original publications listed above.

1. Initialize a starting point for the variational inference algorithm in the $\vec{\xi}$ space. This starting point will be the mean of the initial Gaussian approximation. Typically, a Gaussian random sample is used as the starting point.
2. Draw samples $\vec{\xi}_0, \ldots, \vec{\xi}_n$ from the current Gaussian approximation of the posterior.
3. Use these samples to approximate the Kullback-Leibler divergence between the approximating distribution and the true posterior, and minimize this divergence with respect to the mean of the approximation.
4. Draw new samples from the updated approximation and repeat the minimization of the Kullback-Leibler divergence. Iterate this process until convergence.
5. Output the final set of samples from the approximate distribution to the user.

2.3 NIFTy implementation

In the following, we will discuss how to use the variational inference algorithms presented above in NIFTy. For simplicity, we demonstrate the use of MGVI here. For examples utilizing the more accurate geoVI algorithm, see the demos folder of NIFTy. More advanced features of the `jft.optimize_kl` function, which is used to run the variational inference, can be found on the API reference page.

First, we generate a random starting position for the variational inference run. This random position serves as the mean value of the Gaussian in the initial variational inference iteration. To obtain this random position, we first generate a JAX random number key, which is then used to create a random position in the latent space of our model. Finally, we convert this latent position into a `jft.Vector`. This conversion is necessary because NIFTy needs to perform arithmetic operations on the latent space position. While mathematical operations such as $\vec{\xi}_1 + \vec{\xi}_2$ are defined on latent space vectors, Python cannot add two dictionaries. This issue is resolved by converting the dictionary into a `jft.Vector`.

```
key, subkey = random.split(key, 2)
init_pos = lh.init(subkey)
print("initial position: ", init_pos)
init_pos = jft.Vector(init_pos)
print("initial position vector: ", init_pos)
```

```
initial position:  {'a_input': Array(-0.44370261, dtype=float64), 'b_input':␣
↪Array(-1.93703562, dtype=float64)}
initial position vector:  Vector(
        {'a_input': Array(-0.44370261, dtype=float64),
         'b_input': Array(-1.93703562, dtype=float64)}
)
```

Next, we specify how many iterations of drawing samples and minimizing the Kullback-Leibler divergence we want to do:

```
n_vi_iterations = 6
```

Furthermore, we specify that the number of independent samples used to approximate the Kullback-Leibler divergence should be 4. Note that NIFTy draws pairs of antithetical samples, meaning we will actually have $4 \cdot 2 = 8$ samples.

```
n_samples = 4
```

Drawing random samples from the approximating distribution requires generating JAX random numbers. To do so, we as always need to generate a new JAX random key:

```
key, sampling_key = random.split(key, 2)
```

It is not possible to directly draw samples from the Gaussian posterior approximation due to subtleties in the parametrization of the Gaussian covariance. Therefore, sampling from the current approximation requires not only random numbers but also a numerical optimization algorithm—specifically, the conjugate gradient (CG) algorithm.

For the CG algorithm, we specify the parameter `cg_name=None` to suppress NIFTy's output, making the notebook more readable when rendered as a webpage. In general, however, it is recommended to set `cg_name` to a descriptive string to see the output of the optimizer. Additionally, we pass `cg_kwargs`, where we define a convergence criterion `absdelta=1e-5 * jft.size(lh.domain)` and a maximum number of iterations `maxiter=100` for the CG algorithm.

```
draw_linear_kwargs = dict(
    cg_name=None,
    cg_kwargs=dict(absdelta=1e-5 * jft.size(lh.domain), maxiter=100),
)
```

Similar to sampling, the minimization of the Kullback-Leibler divergence between the true posterior and the approximation is performed using a numerical optimization algorithm, in this case, the Newton conjugate gradient scheme. As before, we set `name=None` to suppress output and make the notebook more readable when viewed as a webpage. For production runs, however, we recommend setting a descriptive string for `name`. Additionally, we define a convergence criterion `xtol=1e-4` and set the maximum number of iterations to `maxiter=35`.

```
kl_kwargs = dict(minimize_kwargs=dict(name=None, xtol=1e-4, maxiter=35))
```

As a final step, we specify the `sampling_mode`. This argument allows the user to choose which variational inference algorithm to use. Here, we set it to `"linear_resample"`, which corresponds to the MGVI algorithm described in arXiv:1901.11033. Other available options can be found on the API reference page.

```
sample_mode = "linear_resample"
```

After having specified all necessary arguments for the variational inference algorithm, we collect these arguments in a Python dictionary.

```
optimize_kl_args = dict(
    likelihood=lh,
    position_or_samples=init_pos,
    n_total_iterations=n_vi_iterations,
    n_samples=n_samples,
    key=sampling_key,
    draw_linear_kwargs=draw_linear_kwargs,
    kl_kwargs=kl_kwargs,
    sample_mode=sample_mode,
)
```

Now we can run the variational inference using the `jft.optimize_kl` function. After each iteration, NIFTy prints a summary that includes:

- **Iteration information**: The current iteration index of the overall variational inference loop, along with the latest estimate of the Kullback-Leibler divergence between the approximation and the true posterior.
- **Sampling details**: For each sample pair (4 in our example), NIFTy reports whether the conjugate gradient method used for sampling was successful. Here, 0 means success and 1 means failure.
- **KL minimization details**: The number of KL minimization steps performed.
- **Results for prior and likelihood**: This section is the most important. NIFTy reports the reduced χ^2 of the residuals between the reconstructed data and the actual data together with the average of the reconstructed data samples and the number of degrees of freedom. A reduced χ^2 value larger than 1 can indicate that the reconstruction has not converged, that noise statistics assumptions are incorrect, or that the model does not fully capture all relevant effects. NIFTy also reports those values for the latent parameters. Since we assume a standard Gaussian prior in the latent space, χ^2 values significantly greater than 1 indicate tension between the prior model and the likelihood.

```
samples, state = jft.optimize_kl(**optimize_kl_args)
```

```
OPTIMIZE_KL: Starting 0001
```

```
CG: gamma=0, converged!
```

```
OPTIMIZE_KL: Iteration 0001 E:+9.3592e+00
OPTIMIZE_KL: Linear sampling status (0, 0, 0, 0)
OPTIMIZE_KL: #(KL minimization steps) 10
OPTIMIZE_KL: Likelihood residual(s):
'reduced Chi²:    0.71±   0.053, avg:    -0.21±    0.12, #dof:     20'
```

(continues on next page)

(continued from previous page)

```
OPTIMIZE_KL: Prior residual(s):
a_input                  :: 'reduced Chi²:     2.8±    0.93, avg:     -1.6±    0.28,
 ↪ #dof:      1'
b_input                  :: 'reduced Chi²:     1.7±     0.9, avg:     -1.3±    0.35,
 ↪ #dof:      1'
```

```
OPTIMIZE_KL: Starting 0002
```

```
OPTIMIZE_KL: Iteration 0002 E:+9.7781e+00
OPTIMIZE_KL: Linear sampling status (0, 0, 0, 0)
OPTIMIZE_KL: #(KL minimization steps) 5
OPTIMIZE_KL: Likelihood residual(s):
'reduced Chi²:    0.72±   0.052, avg:    -0.21±    0.12, #dof:     20'

OPTIMIZE_KL: Prior residual(s):
a_input                  :: 'reduced Chi²:     2.9±     1.5, avg:     -1.6±    0.44,
 ↪ #dof:      1'
b_input                  :: 'reduced Chi²:     2.3±     2.1, avg:     -1.3±    0.77,
 ↪ #dof:      1'
```

```
OPTIMIZE_KL: Starting 0003
```

```
OPTIMIZE_KL: Iteration 0003 E:+1.0518e+01
OPTIMIZE_KL: Linear sampling status (0, 0, 0, 0)
OPTIMIZE_KL: #(KL minimization steps) 9
OPTIMIZE_KL: Likelihood residual(s):
'reduced Chi²:    0.78±    0.16, avg:    -0.21±    0.29, #dof:     20'

OPTIMIZE_KL: Prior residual(s):
a_input                  :: 'reduced Chi²:     3.1±     1.4, avg:     -1.7±    0.41,
 ↪ #dof:      1'
b_input                  :: 'reduced Chi²:     2.3±     2.4, avg:     -1.2±    0.88,
 ↪ #dof:      1'
```

```
OPTIMIZE_KL: Starting 0004
```

```
OPTIMIZE_KL: Iteration 0004 E:+1.0149e+01
OPTIMIZE_KL: Linear sampling status (0, 0, 0, 0)
OPTIMIZE_KL: #(KL minimization steps) 8
OPTIMIZE_KL: Likelihood residual(s):
'reduced Chi²:    0.76±     0.1, avg:    -0.21±    0.24, #dof:     20'

OPTIMIZE_KL: Prior residual(s):
a_input                  :: 'reduced Chi²:     2.9±     1.3, avg:     -1.7±    0.38,
 ↪ #dof:      1'
b_input                  :: 'reduced Chi²:     2.2±     2.1, avg:     -1.3±    0.81,
 ↪ #dof:      1'
```

```
OPTIMIZE_KL: Starting 0005
```

```
OPTIMIZE_KL: Iteration 0005 E:+1.0945e+01
OPTIMIZE_KL: Linear sampling status (0, 0, 0, 0)
OPTIMIZE_KL: #(KL minimization steps) 12
```

(continues on next page)

(continued from previous page)

```
OPTIMIZE_KL: Likelihood residual(s):
'reduced Chi²:     0.8±    0.21, avg:    -0.19±    0.32, #dof:     20'

OPTIMIZE_KL: Prior residual(s):
a_input                 :: 'reduced Chi²:     4.2±     2.6, avg:     -2.0±    0.65,
 ↪ #dof:      1'
b_input                 :: 'reduced Chi²:     1.6±     1.4, avg:     -1.1±    0.59,
 ↪ #dof:      1'
```

```
OPTIMIZE_KL: Starting 0006
```

```
OPTIMIZE_KL: Iteration 0006 E:+1.0071e+01
OPTIMIZE_KL: Linear sampling status (0, 0, 0, 0)
OPTIMIZE_KL: #(KL minimization steps) 9
OPTIMIZE_KL: Likelihood residual(s):
'reduced Chi²:    0.73±     0.1, avg:     -0.2±    0.15, #dof:     20'

OPTIMIZE_KL: Prior residual(s):
a_input                 :: 'reduced Chi²:     3.9±     2.6, avg:     -1.9±    0.67,
 ↪ #dof:      1'
b_input                 :: 'reduced Chi²:     1.6±     1.1, avg:     -1.2±    0.45,
 ↪ #dof:      1'
```

The `optimize_kl` function returns `samples` and `state`. In most use cases, only the `samples` object is relevant as it contains the samples of the approximating distribution from the final iteration. These samples are samples in the latent ξ space. To obtain samples for the quantity of interest, in NIFTy language the signal s, or in our examples the parameters a and b, these samples need to be transformed using the standardized model.

In our examples, this transformation can be done as follows:

```
a_samples = tuple(a(s) for s in samples)
b_samples = tuple(b(s) for s in samples)
y_samples = tuple(my_model(s) for s in samples)
```

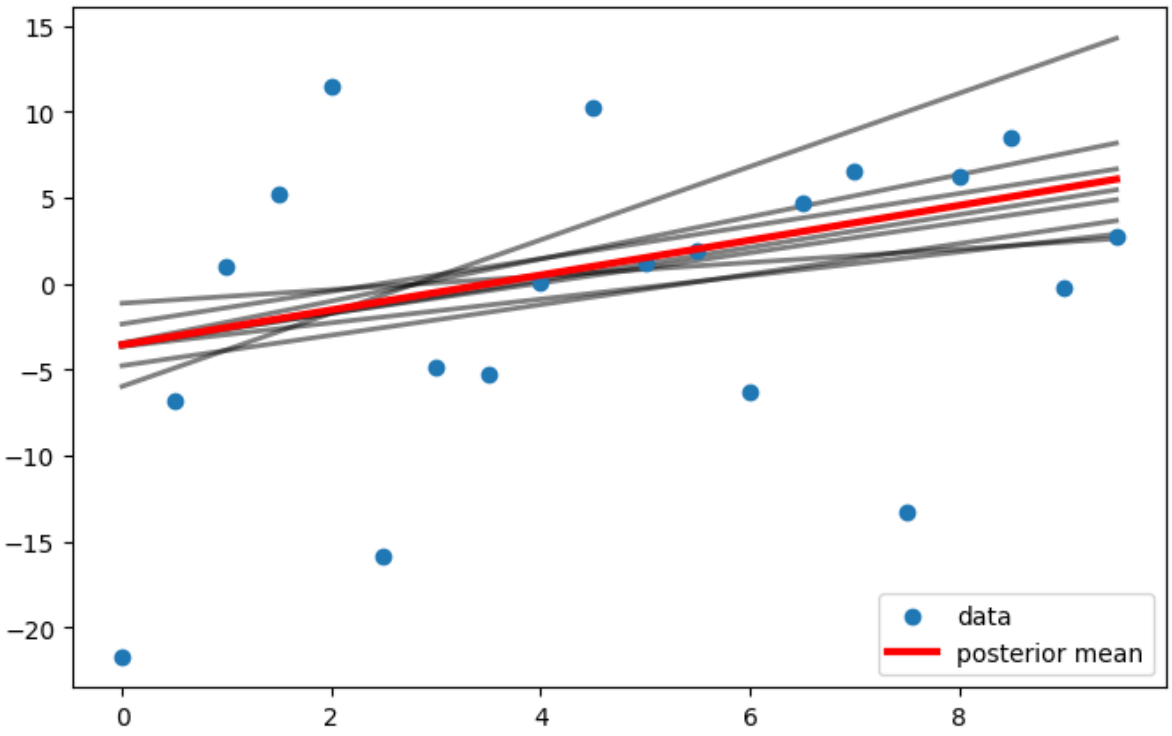
Now we can compute the posterior mean values of our parameters and print them.

```
a_mean = jft.mean(a_samples)
b_mean = jft.mean(b_samples)
y_mean = jft.mean(y_samples)
print("mean of a: ", a_mean)
print("mean of b: ", b_mean)
```

```
mean of a:  1.0142032969059598
mean of b:  -3.5638935790348594
```

Often, it is insightful to plot the reconstruction alongside the data. Such a plot can also be very helpful to detect potential problems in the inference. The plot below visualizes again the data as a scatter plot. On top, the posterior mean of the reconstruction is plotted in red. In grey, the individual samples are shown.

```
plt.figure(figsize=(8, 5))
plt.scatter(x, d, label="data")
plt.plot(x, y_mean, label="posterior mean", linewidth=3.0, color="red")
for ys in y_samples:
    plt.plot(x, ys, linewidth=2.0, color="black", alpha=0.5, zorder=0)
plt.legend()
plt.show()
```



2.4 Summary

This notebook introduced the concept of variational inference. Variational inference is a technique for accessing information from the posterior distribution in a computationally efficient way. The core idea is to approximate the potentially complex posterior distribution with a simpler distribution and generate samples from this simpler distribution. In addition to providing a brief introduction to these concepts, this notebook demonstrates how to use variational inference in NIFTy.

CHAPTER

THREE

GAUSSIAN PROCESSES WITH A FIXED KERNEL

This notebook introduces Gaussian Processes and their implementation in NIFTy. Gaussian Processes are a powerful tool for modeling smooth functions. In the context of NIFTy, we commonly use Gaussian Processes as a prior model. The Gaussian Process prior encodes the assumption that smooth functions are a priori more likely than functions with wild variations. If you are not familiar with NIFTy, please read the notebooks *Models* and *Inference* first.

Before starting with the actual notebook, let's import NIFTy and the relevant libraries.

```
import nifty.re as jft

import jax
import jax.numpy as jnp
import jax.random as random

%matplotlib inline
import matplotlib.pyplot as plt

plt.rcParams["figure.dpi"] = 100

jax.config.update("jax_enable_x64", True)
seed = 42
key = random.PRNGKey(seed)
```

3.1 Motivation

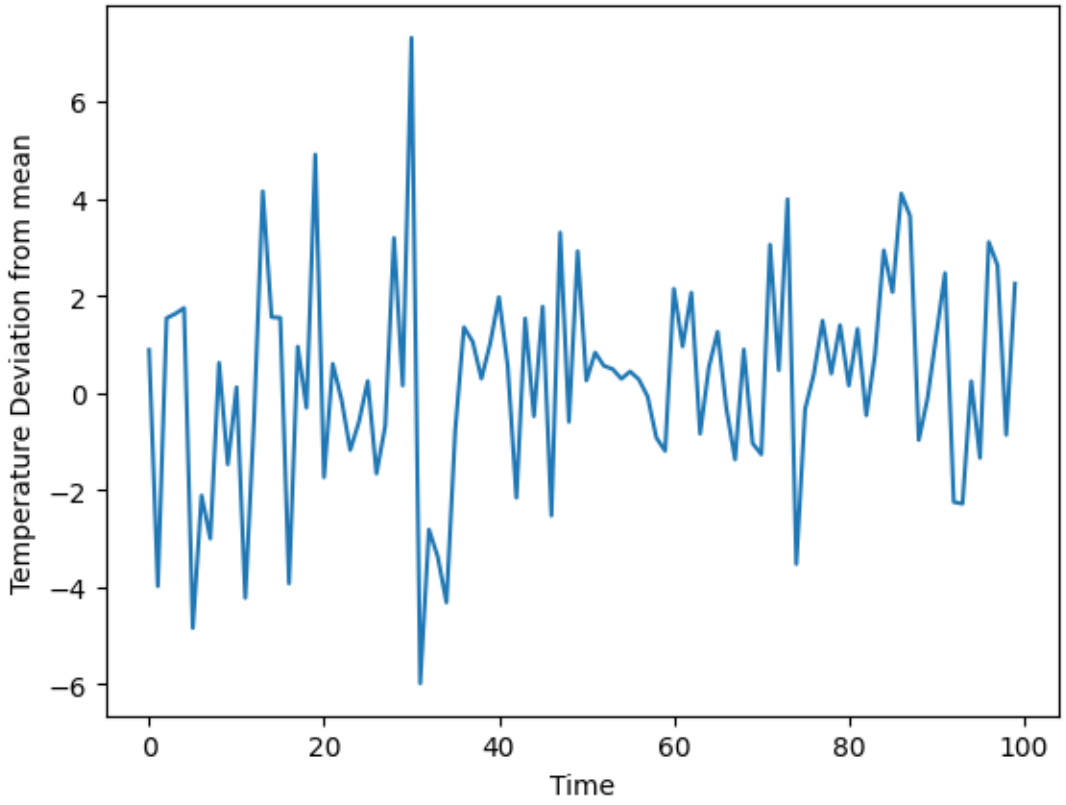
As with many concepts, Gaussian Processes are best introduced with an example. In this notebook, let's consider modeling the temperature in a room as a function of time. We assume that the average temperature in the room is 21 degrees and that it fluctuates with a standard deviation of 2 degrees. Our goal is to model these fluctuations around 21 degrees.In this example, the fluctuations of the temperature around 21 degrees will be our signal s, which we want to reconstruct. The simplest statistical model for s, encoding this prior knowledge, would be a Gaussian distribution with a mean of 0 and a standard deviation of 2.

The first notebook `0_models.py` already introduced a prior model for Gaussian random numbers in NIFTy, namely `jft.NormalPrior`. While `0_models.py` used this model for a single number, we now need a model for multiple numbers, specifically the temperature at different time steps. Using the `shape` argument, multiple random numbers can be generated at once. Assuming we want to model the temperature at 100 time steps, the code below implements the prior model for a vector of 100 Gaussian random numbers with a mean of 0 and a standard deviation of 2.

```
s = jft.NormalPrior(mean=0, std=2, shape=(100))
```

Let's draw a sample of this prior model and visualize it with matplotlib.

```
key, subkey = random.split(key)
latent_sample = jft.random_like(subkey, s.domain)
sample = s(latent_sample)

plt.plot(sample)
plt.xlabel("Time")
plt.ylabel("Temperature Deviation from mean")
plt.show()
```



As expected, the values of the `s` sample fluctuate around 0 with a standard deviation of about 2.

Neighboring entries in the random sample from s are uncorrelated. Thus, in this prior sample, the temperature deviations from 21 degrees jump wildly from one time step to the next. Expressed as a formula, this means that the correlation between the temperature deviations at time steps i and j is zero, $\langle s_x s_y \rangle = 0$ if $x \neq y$. If the neighboring time steps are close to each other, this is unphysical. From physical considerations, we expect the temperature to change gradually as a function of time. Thus, we expect that the values of the temperature at nearby time steps are strongly correlated ($\langle s_x x_y \rangle > 0$ for $x \approx y$). A technique to model such correlations between neighboring points and impose them as a prior on possible functions is the use of Gaussian Processes. In the next section, we will introduce the mathematical background of Gaussian Processes before showcasing their implementation in NIFTy.

3.2 Mathematical background

In the code above, we modeled each time step s_i with an a priori independent Gaussian random variable ξ_i, known as a latent space parameter. Instead of thinking of s as a collection of independent Gaussian random variables, we can also view s as a whole as a multivariate Gaussian random variable. The probability density function of the multivariate normal distribution for the vector s is given by

$$\mathcal{G}(s, S) = \frac{1}{\sqrt{|2\pi S|}} \exp\left(-\frac{1}{2} s^\dagger S^{-1} s\right),$$

where S is the covariance matrix. In the code above, S was a diagonal matrix with $\langle s_i s_i \rangle = 2^2 = 4$ on the diagonal, while the off-diagonal elements, which encode correlations between time steps, were zero. To include correlations between time steps in the prior model, we need to develop a generative model that samples from a multivariate Gaussian with a non-diagonal covariance matrix.

The most straightforward generative model for s is to multiply the latent space standard normally distributed vector ξ with the matrix square root of S:

$$s = \sqrt{S}\, \xi.$$

We can easily verify that an s generated by this procedure has the covariance S:

$$\langle s s^\dagger \rangle = \left\langle \sqrt{S} \xi \xi^\dagger \sqrt{S}^\dagger \right\rangle = \sqrt{S} \left\langle \xi \xi^\dagger \right\rangle \sqrt{S}^\dagger = \sqrt{S} \mathbb{1} \sqrt{S}^\dagger = S.$$

In the second-to-last step, we used the fact that the covariance of the latent parameter vector ξ is the identity matrix, since all latent parameters are a priori independent and standard normally distributed.

The covariance S is an $n \times n$ square matrix, where n is the number of entries in the vector s. In our example, $n = 100$, so S is a 100×100 matrix. However, NIFTy is designed to work with much larger parameter spaces. For example, astronomical imaging applications often deal with images containing a million or more pixels. If s has a million entries, then S would have $(10^6)^2 = 10^{12}$ entries. This is computationally infeasible, as 10^{12} floating-point numbers with 64-bit precision would require 8 terabytes of memory. Therefore, to enable applications with large parameter spaces, we need to circumvent the quadratic scaling and cannot directly work with S or $\sqrt{S}$.

3.2.1 Wiener-Khintchin Theorem

To circumvent the quadratic scaling, we a priori assume statistical homogeneity and isotropy of the signal s, meaning translation invariance and directional independence of the covariance matrix S. In terms of our example, this means that we assume a priori that the correlation between temperature fluctuations at two time steps depends only on the distance between them. Expressed as a formula, the entries of the covariance matrix depend only on the distance between the time steps:

$$S_{xy} = \langle s_x s_y \rangle = C_s(|x - y|).$$

Under the assumptions of statistical homogeneity and isotropy, the Wiener–Khinchin theorem states that the covariance becomes diagonal in harmonic space:

$$\tilde{S}_{kk'} = \left\langle s_k s_{k'}^\dagger \right\rangle = 2\pi \delta(k - k') P(|k|),$$

where $\tilde{S}$ is the harmonic space covariance matrix, and the diagonal is populated by the one-dimensional power spectrum.

3.2.2 NIFTy Gaussian Process Model

The assumptions of statistical homogeneity and isotropy, combined with the Wiener–Khinchin theorem, are the central ingredients of the Gaussian Process model in NIFTy. Through the Wiener–Khinchin theorem, the covariance matrix can be fully represented by the power spectrum, avoiding quadratic scaling. To map latent independent standard Gaussian random variables $\vec{\xi}$ to a multivariate Gaussian random variable $\vec{s}$, NIFTy uses the following procedure. It starts with a standard normal random vector $\vec{\xi}$, where each entry in the vector corresponds to one harmonic frequency k of the signal we want to generate. Next, NIFTy multiplies $\vec{\xi}$ by the square root of the harmonic space covariance matrix $\sqrt{\tilde{S}_{kk'}}$ and then applies a harmonic transform to the result. Expressed as a formula, the procedure is:

$$s = \mathrm{HT}\sqrt{\tilde{S}}\,\xi.$$

The square root of $S_{kk'}$ can easily be computed since $S_{kk'}$ is a diagonal matrix. We can verify that the covariance of s is indeed S as desired:

$$\left\langle s_x s^\dagger_{x'} \right\rangle = \left\langle \mathrm{HT}\sqrt{\tilde{S}}\xi\xi^\dagger\sqrt{\tilde{S}}^\dagger\,\mathrm{HT}^\dagger \right\rangle = \mathrm{HT}\sqrt{\tilde{S}}\left\langle \xi\xi^\dagger\right\rangle\sqrt{\tilde{S}}^\dagger\,\mathrm{HT}^\dagger = \mathrm{HT}\sqrt{\tilde{S}}\mathbb{1}\sqrt{\tilde{S}}^\dagger\,\mathrm{HT}^\dagger = S.$$

This is the idea of the fixed power spectrum Gaussian process model in NIFTy. In the remainder of the notebook, we will explain the implementation of this model.

3.3 Implementation in NIFTy

We begin the implementation by defining the grid on which our signal lives. For this, we specify `dims=100` to indicate that we have 100 time steps. Furthermore, we specify the distance between the time steps with `distances = 0.01`. In this example, the units of the time distances are arbitrary. However, in your applications, you should, of course, always be aware of the units.

```
dims = 100
distances = 0.01
```

With this information on the number of pixels and the distances between them, we now initialize a NIFTy grid containing this information. To do this, we use a utility function from `jft.correlated_field`. `jft.correlated_field` is a model for generating Gaussian processes with unknown power spectra, which you will learn more about in later notebooks. Here, we focus on generating a Gaussian process model with a known power spectrum.

To initialize the grid, we pass to the `jft.correlated_field.make_grid` function, besides `dims` and `distances`, information about the harmonic counterpart of the space in which our signal lives. This is necessary because we need to perform a harmonic transformation to generate a Gaussian process via the Wiener–Khinchin theorem. Here, we set `harmonic_type="fourier"`. If our signal lived on a sphere instead of a regular Cartesian grid, we would set it to `"spherical"`.

```
grid = jft.correlated_field.make_grid(
    dims, distances=distances, harmonic_type="fourier"
)
```

Next, we define a Python function for the power spectrum—or more precisely, for the square root of the power spectrum, which we refer to as the amplitude spectrum. The `amplitude_spectrum` function takes as input the length of the harmonic factor `k` and evaluates the amplitude spectrum for this k-mode.

```
def amplitude_spectrum(k):
    return 2.5 / (5 + k**2)
```

After defining the Python function for the power spectrum, we can initialize the array containing the square root of the signal covariance in harmonic space. To do this, we first evaluate the amplitude spectrum function at all lengths of the Fourier vectors in our grid. The grid property `grid.harmonic_grid.mode_lengths` returns a vector containing all unique lengths of k-vectors in our Fourier grid. We evaluate the amplitude spectrum function at these k-vector lengths.

```
k_lengths = grid.harmonic_grid.mode_lengths
amplitudes = amplitude_spectrum(k_lengths)
print("k_length: ", k_lengths)
print("amplitudes: ", amplitudes)
```

```
k_length:  [ 0.  1.  2.  3.  4.  5.  6.  7.  8.  9. 10. 11. 12. 13. 14. 15. 16. 17.
 18. 19. 20. 21. 22. 23. 24. 25. 26. 27. 28. 29. 30. 31. 32. 33. 34. 35.
 36. 37. 38. 39. 40. 41. 42. 43. 44. 45. 46. 47. 48. 49. 50.]
amplitudes:  [0.5        0.41666667 0.27777778 0.17857143 0.11904762 0.08333333
 0.06097561 0.0462963  0.03623188 0.02906977 0.02380952 0.01984127
 0.01677852 0.01436782 0.01243781 0.01086957 0.00957854 0.0085034
 0.00759878 0.0068306  0.00617284 0.00560538 0.00511247 0.00468165
 0.00430293 0.00396825 0.00367107 0.00340599 0.00316857 0.00295508
 0.00276243 0.00258799 0.00242954 0.00228519 0.00215332 0.00203252
 0.0019216  0.00181951 0.00172533 0.00163827 0.00155763 0.0014828
 0.00141323 0.00134844 0.001288   0.00123153 0.00117869 0.00112918
 0.00108272 0.00103907 0.000998  ]
```

We now have the corresponding amplitudes for all unique k-vector lengths in our grid. From these amplitudes, we need to construct the diagonal of the Fourier space representation of the square root of the covariance matrix. This is somewhat tedious, as multiple Fourier vectors can share the same length. In our 1D example, there is the zero mode—i.e., the k-vector with length 0—but then there are two next longer k-vectors: $k = 0.01$ and $k = -0.01$. In higher-dimensional spaces, even more Fourier vectors share the same length.

To simplify mapping from the amplitude array to the diagonal of the harmonic space square root covariance matrix, the `grid` contains the property `grid.harmonic_grid.power_distributor`. The `power_distributor` is an array containing, for each element of the diagonal of the harmonic space covariance matrix, the index of the corresponding k-vector length in the `grid.harmonic_grid.mode_lengths` array. Using this index array, we can easily obtain the diagonal of the square root covariance from the amplitude array.

```
sqrt_harmonic_cov = amplitudes[grid.harmonic_grid.power_distributor]
```

Now we have all the components to initialize a NIFTy model that transforms a standard normally distributed ξ into a multivariate Gaussian with the power spectrum we defined. To the constructor of `FixedPowerCorrelatedField`, we pass `sqrt_harmonic_cov` and the `grid`. Inside the constructor, we store the harmonic transformation `ht` as an attribute of the class. For the harmonic transformation, we use the `jft.correlated_field.hartley` function. The Hartley transformation is closely related to the Fourier transformation, but for our application, it has the advantage of transforming real-valued functions to real-valued functions. To avoid dealing with complex numbers, we use the Hartley transformation instead of a normal Fourier transformation. Furthermore, we store a volume scaling factor `harmonic_dvol` as a property. In the `__call__` method, we scale the result of the Hartley transformation with this factor to comply with the NIFTy convention.

The `__call__` method performs the mapping from the latent parameters to the Gaussian process realizations. Following the formulas in the mathematical background section, we multiply the input vector by the square root of the harmonic space covariance matrix `sqrt_harmonic_cov` and then apply a harmonic transformation to the result.

```
class FixedPowerCorrelatedField(jft.Model):
    def __init__(self, sqrt_harmonic_cov, grid):
        self.sqrt_harmonic_cov = sqrt_harmonic_cov
        self.ht = jft.correlated_field.hartley
        self.harmonic_dvol = 1 / grid.total_volume
```

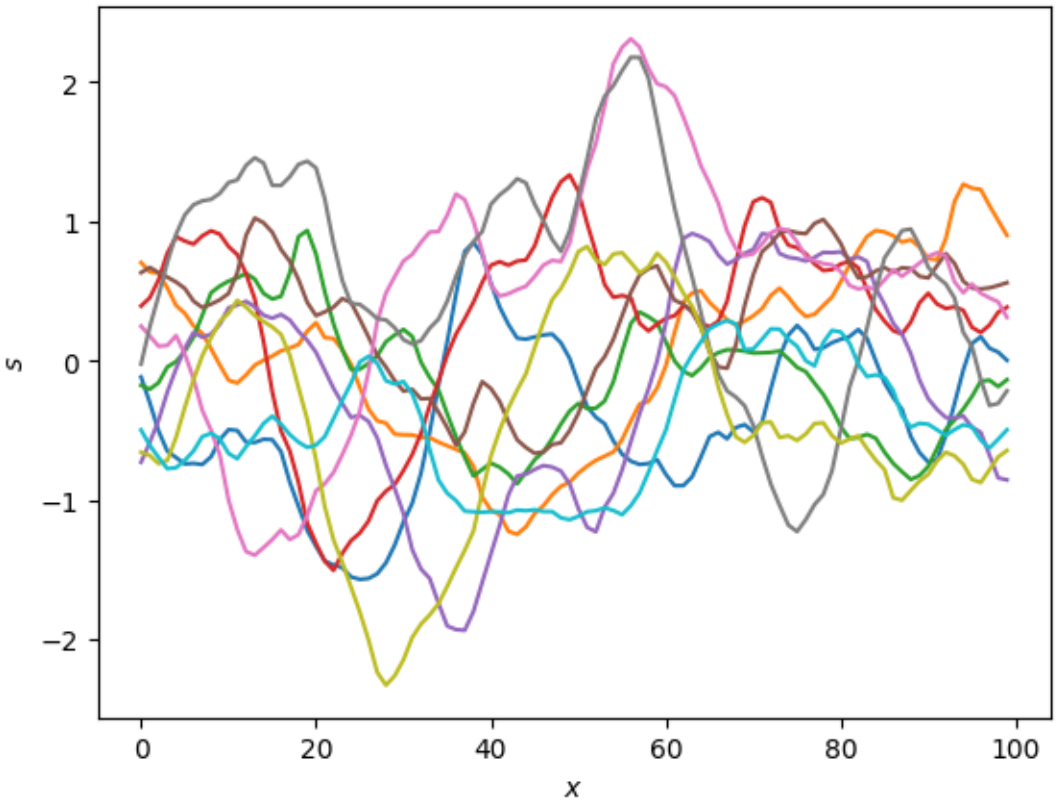
(continues on next page)

(continued from previous page)

```
        super().__init__(
            domain=jax.ShapeDtypeStruct(shape=grid.shape, dtype=jnp.float64)
        )

    def __call__(self, x):
        return self.harmonic_dvol * self.ht(self.sqrt_harmonic_cov * x)


s = FixedPowerCorrelatedField(sqrt_harmonic_cov, grid)
```

To visualize our fixed-power spectrum Gaussian process model, we draw 10 standard normal distributed input vectors, map them through our model to a correlated multivariate Gaussian sample, and plot the results. As expected, the samples are now no longer wildly jumping from one time step to the next, but smoothly fluctuate.

```
for i in range(10):
    key, subkey = random.split(key)
    latent_sample = jft.random_like(subkey, s.domain)
    sample = s(latent_sample)
    plt.plot(sample)
    plt.xlabel(r"$x$")
    plt.ylabel(r"$s$")
plt.show()
```



3.4 Summary

This notebook introduced a model for Gaussian processes and its implementation in `NIFTy`. By utilizing the Wiener–Khinchin theorem and the assumptions of a priori statistical homogeneity and isotropy, the implementation does not explicitly parameterize the covariance matrix of the Gaussian process or any other quantity that scales quadratically with the number of pixels. This makes the model applicable even in settings with a large number of pixels. In the next notebook, *Wiener Filter*, we will continue with the Gaussian process model presented here and showcase some reconstructions. Thereby, we will introduce an alternative to the variational inference algorithm presented in the *inference notebook*.

The Gaussian process model presented in this notebook assumes that the power spectrum of the signal is known. However, in many real-world applications, this is not the case. A more flexible model applicable to settings where the underlying power spectrum is unknown will be presented in *this notebook*.

CHAPTER

FOUR

WIENER FILTER

NIFTy primarily uses Variational Inference to approximate posterior distributions. However, for linear models with Gaussian likelihoods, variational inference is not required, as the posterior can be computed directly for this special case. The formula for directly computing the posterior of a linear model is called the Wiener Filter. NIFTy also implements the Wiener Filter, and this notebook demonstrates how to use it. It provides a brief introduction to the mathematical background of the Wiener Filter. For a complete derivation, see, for example, this script.

Before we start with the Wiener filter implementation in `NIFTy.re`, let us quickly review the mathematical background.

The Wiener Filter makes the following assumptions:

- A linear measurement response R relating the signal s to the measured data $d = Rs + n$ and the noise n.
- A Gaussian prior on the signal s. For the simple Wiener Filter, the Gaussian needs to have mean 0. The covariance of the Gaussian prior is denoted by S, i.e., $\mathcal{P}(s) = \mathcal{G}(s, S)$.
- Analogously, the noise n also needs to be Gaussian distributed: $\mathcal{P}(n) = \mathcal{G}(n, N)$.

As a consequence of the Gaussian noise, the likelihood is also a Gaussian distribution:

$$\mathcal{P}(d|s) = \mathcal{G}(d - Rs, N).$$

The joint distribution $\mathcal{P}(s, d)$, which is the product of the likelihood and the prior, can be written as:

$$\mathcal{P}(s, d) = \mathcal{P}(d|s)\,\mathcal{P}(s) = \mathcal{G}(d - Rs, N)\mathcal{G}(s, S).$$

The product of two Gaussian distributions remains Gaussian, so the posterior is also a Gaussian distribution. Via direct calculation, it can be shown that the posterior distribution can be written as:

$$\mathcal{P}(s|d) = \frac{\mathcal{P}(s, d)}{\mathcal{P}(d)} = \mathcal{G}(s - m, D),$$

with the posterior covariance being:

$$D^{-1} = S^{-1} + R^\dagger N^{-1} R,$$

and the mean being:

$$m = DR^\dagger N^{-1} d.$$

In some applications, D is called the information propagator and $j = R^\dagger N^{-1} d$ the information source.

To summarize, under the assumptions of the Wiener filter, the posterior is a Gaussian distribution, and the posterior mean m can be computed via $m = F_W d$ with:

$$F_W = DR^\dagger N^{-1} = \left(S^{-1} + R^\dagger N^{-1} R\right)^{-1} R^\dagger N^{-1}.$$

Through a short calculation, it can be shown that the Wiener Filter is equivalent to the optimal linear filter:

$$F_L = SR^\dagger \left(RSR^\dagger + N\right)^{-1},$$

which is designed to minimize the expected root mean square between the signal s and the posterior mean m.

As introduced in previous notebooks, NIFTy builds on generative models mapping from a standard normal distribution to the desired prior signal distribution. For this reason, the prior covariance S is (in `NIFTy.re`) always given by the unit matrix. The generative model mapping the latent parameters to the desired signal prior distribution becomes part of the response R. Thus, the Wiener filter for NIFTy reads:

$$F_W = \left(\mathbb{1} + R^\dagger N^{-1} R\right)^{-1} R^\dagger N^{-1}.$$

4.1 Wiener Filter in NIFTy

To demonstrate the Wiener filter, we will continue the Gaussian process example from the *previous notebook*. To recapitulate, in the previous notebook we coded a generative Gaussian process model of the form:

$$s = \mathrm{HT}\, A\xi,$$

with HT being the Hartley transform and A the amplitude spectrum. Inserting this generative model into the measurement equation, we get:

$$d = R_I s + n = R_I \mathrm{HT}\, A\xi + n = R\xi + n,$$

with R_I being the response of the instrument and R the combination of generative model and instrument response. The following code block contains the generative model from the previous notebook.

```
import nifty.re as jft

import jax
import jax.numpy as jnp
import jax.random as random

%matplotlib inline
import matplotlib.pyplot as plt

plt.rcParams["figure.dpi"] = 100

jax.config.update("jax_enable_x64", True)
seed = 42
key = random.PRNGKey(seed)


dims = 100
distances = 0.01
grid = jft.correlated_field.make_grid(
    dims, distances=distances, harmonic_type="fourier"
)

axis = jnp.arange(0, dims * distances, distances)


def amplitude_spectrum(k):
    return 2.5 / (5 + k**2)
```

(continues on next page)

(continued from previous page)

```
k_lengths = grid.harmonic_grid.mode_lengths
amplitudes = amplitude_spectrum(k_lengths)
sqrt_hamonic_cov = amplitudes[grid.harmonic_grid.power_distributor]


class FixedPowerCorrelatedField(jft.Model):
    def __init__(self, sqrt_hamonic_cov, grid):
        self.sqrt_hamonic_cov = sqrt_hamonic_cov
        self.ht = jft.correlated_field.hartley
        self.harmonic_dvol = 1 / grid.total_volume

        super().__init__(
            domain=jax.ShapeDtypeStruct(shape=grid.shape, dtype=jnp.float64)
        )

    def __call__(self, x):
        return self.harmonic_dvol * self.ht(self.sqrt_hamonic_cov * x)


signal = FixedPowerCorrelatedField(sqrt_hamonic_cov, grid)
```

For an initial example, we will assume a perfect instrument, meaning that we set $R_{\mathrm{I}} = \mathbb{1}$ to the unit matrix. In later examples, we will consider a more complicated response function. Thus, for now, the signal response Rs is equal to s.

```
signal_response = signal
```

4.1.1 Synthetic data generation

In the previous notebooks, we loaded a data file as you would do in an application to real data. In this notebook, we will generate synthetic data by drawing prior samples with the following procedure. First, we draw a standard normal sample on the input domain of our generative model. We name this sample `pos_truth` as it will be the ground truth. We pass this latent space sample through our model to compute the corresponding signal response.

```
key, subkey = random.split(key)
pos_truth = jft.random_like(subkey, signal_response.domain)
signal_response_truth = signal_response(pos_truth)
```

To generate synthetic data consistent with the measurement equation $d = Rs + n$, we draw a random noise sample. For the example in this notebook, we will assume that the noise is uncorrelated between the data points and has a standard deviation of 0.3.

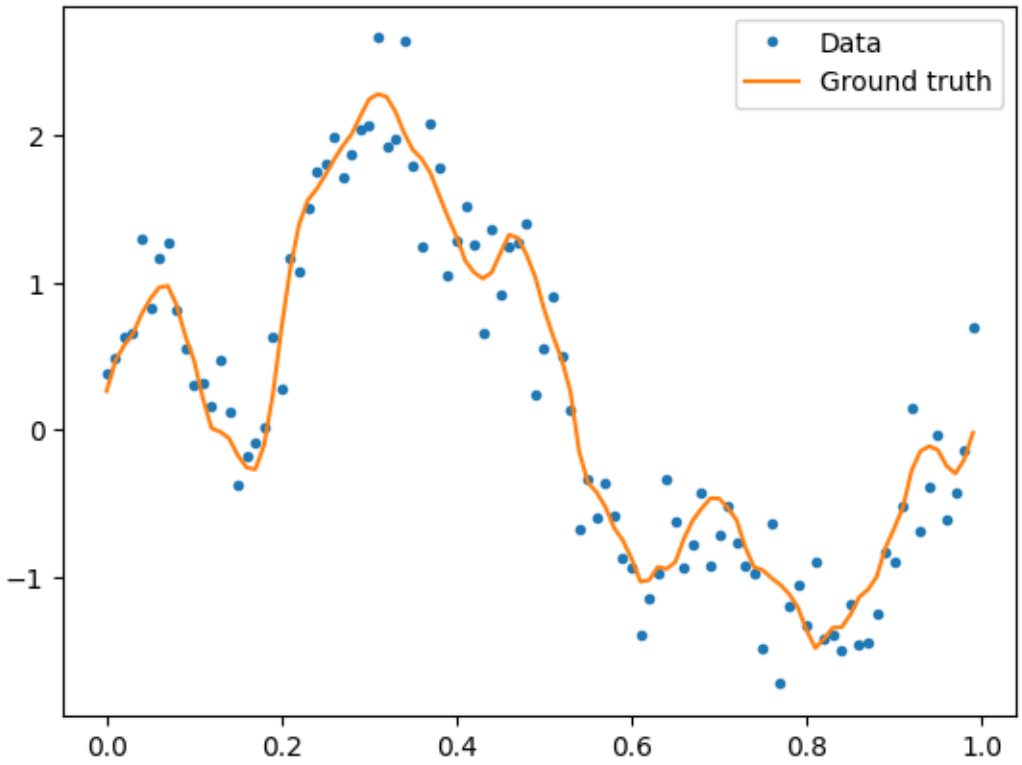
```
noise_std = 0.3
noise_cov = lambda x: noise_std**2 * x
noise_cov_inv = lambda x: noise_std**-2 * x
key, subkey = random.split(key)
noise_truth = noise_std * jft.random_like(key, signal_response.target)
```

Having now random realizations for s and n, we can construct the synthetic data $d = Rs + n$.

```
data = signal_response_truth + noise_truth
```

Let's plot the synthetic data together with the ground truth.

```
plt.plot(axis, data, ".", color="tab:blue", label="Data")
plt.plot(axis, signal(pos_truth), color="tab:orange", label="Ground truth")
plt.legend()
plt.show()
```



`wiener_filter_posterior` does not directly take data as an input parameter, but rather the likelihood connected to the data. As we assume a linear measurement model with Gaussian signal and noise, the likelihood is a Gaussian of the following form:

$$\mathcal{P}(d|s) = \mathcal{G}(d - Rs, N)$$

```
lh = jft.Gaussian(data, noise_cov_inv).amend(signal_response)
```

```
assuming a diagonal covariance matrix;
setting `std_inv` to `cov_inv(ones_like(data))**0.5`
```

```
/usr/local/lib/python3.13/site-packages/nifty/re/model.py:190: UserWarning:␣
 ↪drawing white parameters;
to silence this warning, overload the `init` method
  warn(msg)
```

The following parameters are most important for the `wiener_filter_posterior`. Additional parameters for further options are documented in the API reference.

- `likelihood`: The NIFTy likelihood of the inference problem contains the data, the noise covariance, and the signal response.
- `position`: This parameter is optional and is only important for non-linear models or non-Gaussian likelihoods. If the model is non-linear, the model gets linearized around this position to apply the Wiener filter formulas.

However, for non-linear models, the Wiener filter is no longer exact, and you should consider using variational inference discussed in the *this notebook*.

- `key`: JAX random number generation key used to generate keys for drawing posterior samples.
- `n_samples`: Number of posterior samples to draw.
- `draw_linear_kwargs`: Specifies the parameters used for the conjugate gradient scheme to obtain the Wiener filter solution. For numerical reasons, `NIFTy` does not directly invert D^{-1}, but instead applies D via conjugate gradient minimization. For more information on how to set the parameters of the conjugate gradient, see the *inference notebook*.
- `optimize_for_linear`: Enables numerical optimization for linear models.

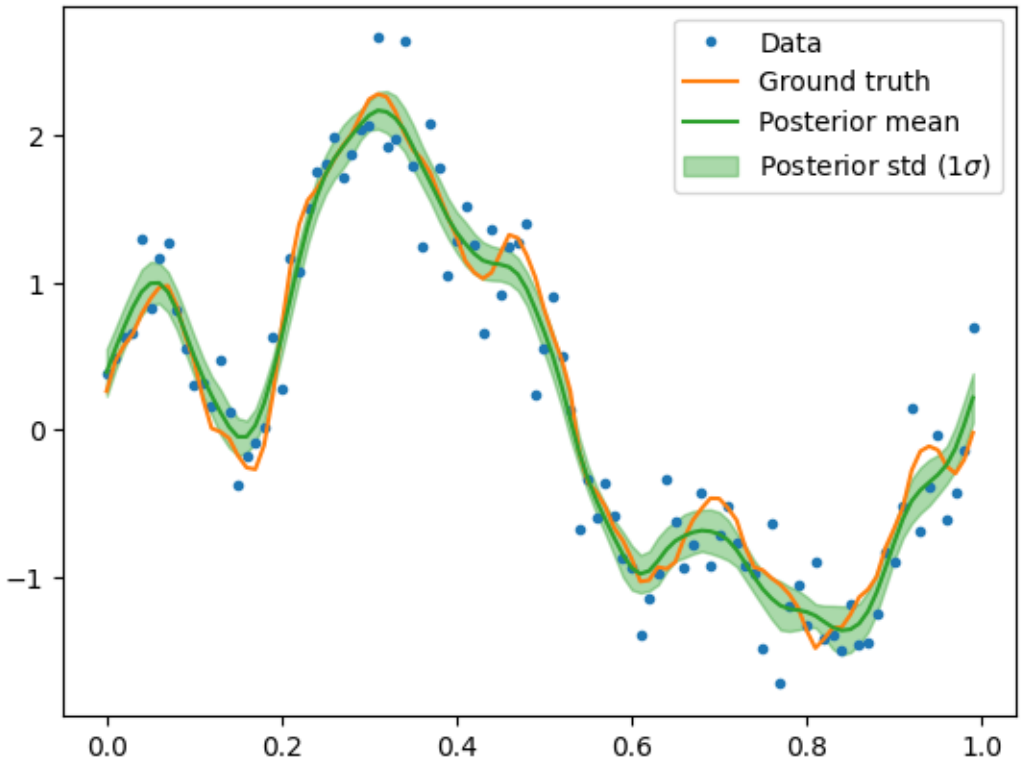
The `wiener_filter_posterior` method returns a set of samples from the posterior distribution. The mean of the samples equals the posterior mean computed via the Wiener filter formulas, as the samples are drawn synthetically around the mean.

```
# The Wiener filter

delta = 1e-6
key, k_w = random.split(key)
samples, _ = jft.wiener_filter_posterior(
    likelihood=lh,
    key=k_w,
    n_samples=20,
    draw_linear_kwargs=dict(
        cg_name=None,
        cg_kwargs=dict(absdelta=delta * jft.size(lh.domain) / 10.0, maxiter=100),
    ),
)
```

With the drawn `samples` around the Wiener filter solution, we can now calculate the posterior statistics, its mean and standard deviation of the `signal`, and compare it to the ground truth.

```
post_mean, post_std = jft.mean_and_std(tuple(signal(s) for s in samples))
```

```
plt.plot(axis, data, ".", color="tab:blue", label="Data")
plt.plot(axis, signal(pos_truth), color="tab:orange", label="Ground truth")
plt.plot(axis, post_mean, label="Posterior mean", color="tab:green")
plt.fill_between(
    axis,
    post_mean - post_std,
    post_mean + post_std,
    color="tab:green",
    alpha=0.4,
    label=r"Posterior std (1$\sigma$)",
)
plt.legend()
plt.show()
```



The plot above shows, besides the data and the ground truth signal, the Wiener filter reconstruction of the `signal` together with its 1σ band. The Wiener filter solution mostly captures the ground truth signal of the synthetic data.

4.2 Wiener Filter on incomplete data

As a second example, let's change the signal response to a more complicated function. The signal response is no longer a unit matrix, but a point-wise multiplication with a sensitivity. For a given slice of the signal array, we set the sensitivity to 0, meaning that the data carries no information about the signal in this region.

In the NIFTy code, we define a class `SignalResponse` to multiply the signal with the `sensitivity` array.

```
class SignalResponse(jft.Model):
    def __init__(self, signal, sensitivity):
        self.signal = signal
        self.sensitivity = sensitivity
        super().__init__(domain=signal.domain)

    def __call__(self, x):
        return self.signal(x) * self.sensitivity


sensitivity = jnp.ones(shape=dims)
sensitivity = sensitivity.at[25:80].set(0.0)
signal_response = SignalResponse(signal, sensitivity)
```

As before, we generate synthetic data consistent with the measurement equation $d = Rs + n$.

```
noise_std = 0.1
noise_cov = lambda x: noise_std**2 * x
noise_cov_inv = lambda x: noise_std**-2 * x

signal_truth = signal(pos_truth)

signal_response_truth = signal_response(pos_truth)
key, subkey = random.split(key)
noise_true = noise_std * jft.random_like(subkey, signal_response.domain)
data = signal_response_truth + noise_true
```

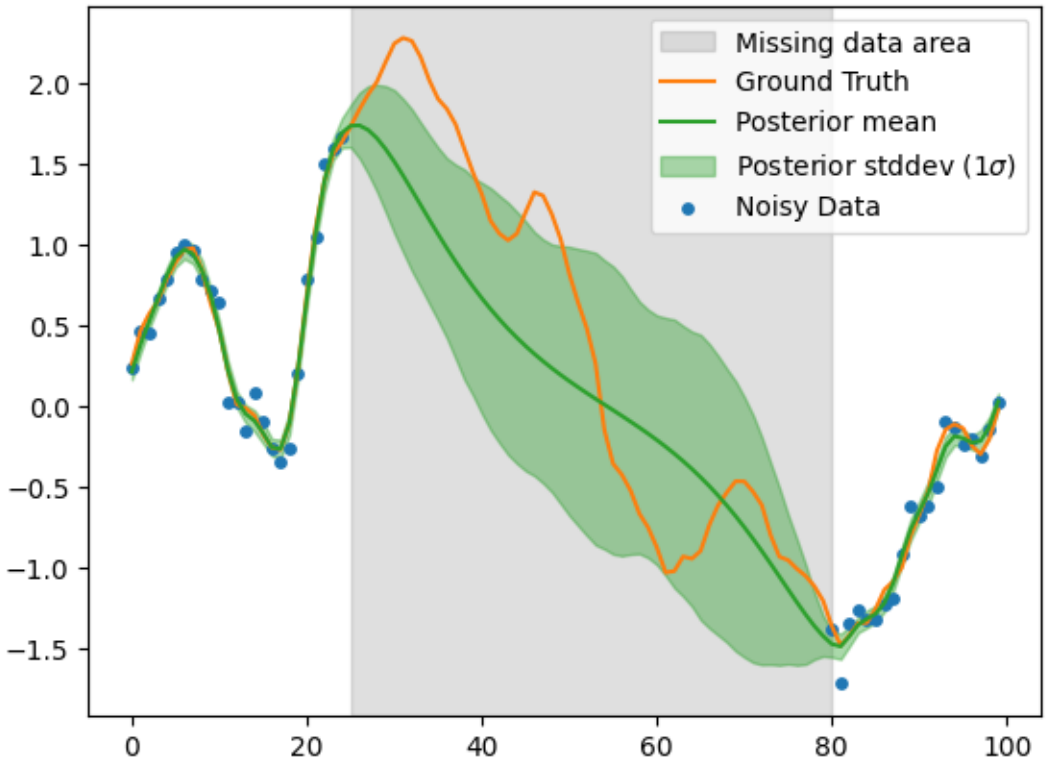
Again, we assume Gaussian noise to construct the likelihood for the problem and call the `wiener_filter_posterior` function. We compute the posterior mean and standard deviation for the reconstructed `signal`.

```
lh = jft.Gaussian(data, noise_cov_inv).amend(signal_response)

key, subkey = random.split(key)
samples, info = jft.wiener_filter_posterior(
    lh,
    key=subkey,
    n_samples=20,
    draw_linear_kwargs=dict(cg_name=None, cg_kwargs=dict(absdelta=delta,␣
 ↪maxiter=100)),
)

post_mean, post_std = jft.mean_and_std(tuple(signal(s) for s in samples))
```

```
assuming a diagonal covariance matrix;
setting `std_inv` to `cov_inv(ones_like(data))**0.5`
```

As before, we plot the reconstruction alongside the ground truth of the signal and the data. Furthermore, we shade the area where the `sensitivity` is zero in grey.

```
plt.axvspan(25, 80, color="gray", alpha=0.25, label="Missing data area")
plt.plot(jnp.arange(len(data)), signal_truth, color="tab:orange", label="Ground Truth
 ↪")
plt.plot(jnp.arange(len(data)), post_mean, color="tab:green", label="Posterior mean")
plt.fill_between(
    range(post_mean.size),
    (post_mean - post_std),
    (post_mean + post_std),
    color="tab:green",
    alpha=0.4,
    label=r"Posterior stddev (1$\sigma$)",
)

data = data.at[25:80].set(jnp.nan)  # Remove masked data from plotting
plt.scatter(jnp.arange(len(data)), data, label="Noisy Data", s=15, color="tab:blue")
plt.legend()
plt.show()
```



Again, we can see that the Wiener filter solution recovers the signal in regions where data is available. We can see that it interpolates in the area where we have no data, given the fixed covariance structure. Furthermore, the Wiener filter solution assigns greater posterior uncertainty to the regions unconstrained by the data.

4.3 Summary

This notebook introduced the Wiener filter and showed how to apply the Wiener filter formulas in `NIFTy.re` using the `wiener_filter_posterior` function. Furthermore, the limitations of the Wiener filter for nonlinear models were highlighted. Additionally, the notebook introduced the idea of generating synthetic data from the prior models. Reconstructing synthetically generated data is not only handy for introductory examples but is also a good test for new models and inference algorithms.

CHAPTER

FIVE

GAUSSIAN PROCESSES WITH A VARIABLE KERNEL

In this notebook we explain the implementation of Gaussian processes with variable power spectra in `NIFTy`. In the previous Gaussian process *notebook* we implemented a generative model for Gaussian processes with a fixed correlation kernel, or fixed power spectrum assuming that it is known. However, for many inference setups, this is not the case and the correlation structures are a priori unknown.

To overcome this limitation, we introduce a Gaussian process model where the exact power spectrum can be a priori unknown. The basic underlying idea is to set up a generative model for power spectra and make this power spectrum model part of the overall generative model for the Gaussian process.

Let us begin with a brief recap of the generative Gaussian process model with a fixed power spectrum from the previous notebook.

5.1 Fixed Power Spectrum Model

In `NIFTy`, we leverage the Wiener-Khinchin theorem to build our Gaussian process models. The Wiener-Khinchin theorem states that the correlation structure for a statistically homogeneous and isotropic signal s can be expressed in harmonic space as a diagonal matrix. This means that the covariance matrix for the signal, S, can be written in harmonic space as:

$$\tilde{S}_{kk'} = \left\langle s_k s^\dagger_{k'} \right\rangle = 2\pi\delta(k-k')P(|k|)$$

where k and k' are the Fourier coefficients, $\tilde{S}_{kk'}$ is the harmonic space covariance matrix, and $P(|k|)$ is the power spectrum.

With the Wiener-Khinchin theorem, modeling the correlation structure of an n-dimensional system is reduced from scaling as $\mathcal{O}(n^2)$ to scaling as $\mathcal{O}(n)$.

A signal s is then modeled starting in harmonic space, as:

$$s = \mathrm{HT}(A\xi)$$

where ξ is a standard normal vector, $\xi \sim \mathcal{N}(0,1)$, A is the amplitude spectrum related to $\tilde{S}$ as $A = \sqrt{\tilde{S}}$, and HT is the harmonic transform.

Since A is fixed in this case, we can only define Gaussian processes with a fixed power spectrum. This narrows the range of the signal realizations we can capture with the model.

To circumvent this constraint and to capture a wider span of signals, we make the amplitude spectrum A a part of the model. This means that we build a generative model for A, which is then applied to the previously introduced latent standard normal random vector.

Expressed as a formula, the idea for a generative Gaussian process model with variable power spectra is:

$$s(\xi_0, \xi_1) = \mathrm{HT}\left(A(\xi_0)\xi_1\right)$$

where now A is a generative model for the amplitude spectrum, which transforms the standard normally distributed latent parameters ξ_0 to possible amplitude spectra. The result $A(\xi_0)$ is then applied to the latent parameters ξ_1, introduced in the previous notebooks. Thus, the variable power spectrum model takes two sets of latent parameters as input, ξ_0 and ξ_1. From ξ_0 an amplitude spectrum is generated, which is then applied to ξ_1.

There are many potential generative models for amplitude spectra $A(\xi_0)$. NIFTy implements two options: first, to parameterize the amplitude spectrum with a Matérn kernel with learnable parameters, and second, to use a non-parametric model for the amplitude spectrum. In the following, we will showcase the non-parametric model.

5.2 Non-parametric correlated field model

Many physical processes lead to power-law-like power spectra. For this reason, the non-parametric amplitude spectrum model of NIFTy builds upon a power law (with variable slope) to which multiplicative non-parametric deviations are added. A set of hyperparameters allows steering the prior distributions on the slope of the power law and other components of the model in order to tailor this general power spectrum model toward specific applications. These hyperparameters are named `loglogavgslope`, `fluctuations`, `offset_mean`, `offset_std`, `flexibility`, and `asperity`.

In this notebook, we provide an intuition for these hyperparameters and instructions on how to set them in your applications. For complete mathematical derivations, we refer to this publication.

Let us begin by importing the required modules and initializing our space to build our prior models on. For this notebook, we will work with a 1-dimensional space with 256 points, and the distance between those points is given as 1, in the appropriate units.

```
import nifty.re as jft
from jax import random
from jax import numpy as jnp
import numpy as np
import jax

%matplotlib inline
import matplotlib.pyplot as plt

plt.rcParams["figure.dpi"] = 100

jax.config.update("jax_enable_x64", True)

shape = 256
distances = 1
seed = 50
key = random.PRNGKey(seed)
```

5.2.1 `CorrelatedFieldMaker` overview

The Gaussian process models in NIFTy with variable amplitude spectra are constructed using the `jft.CorrelatedFieldMaker` helper class. In most NIFTy applications, the `CorrelatedFieldMaker` is directly used to generate a Gaussian process model. For this notebook, we will pack it inside a helper function which we call `fieldmaker`, as we will instantiate several Gaussian process models to visualize how the model works. In the code below you find this helper function. The following list gives a high-level overview of how the `jft.CorrelatedFieldMaker` works before going into the details further below in this notebook.

- First, we instantiate an instance `cfm` of the `jft.CorrelatedFieldMaker`. The initialization method of the correlated field maker gets an argument called `prefix`. This string is used as a key in the dictionary with the

input for the Gaussian process model.

- Second, we call the function `set_amplitude_total_offset` of the `cfm` object. This function takes as arguments `offset_mean` and `offset_std` and sets the mean of the Gaussian process. We will discuss the exact meaning of the two arguments of this function further below.
- As a third step, a model for the fluctuations of the Gaussian process around its mean value is added with `cfm.add_fluctuations`. The `add_fluctuations` function gets several arguments:
- `shape`: This is the shape of the Gaussian process we want to generate. Here we want to generate a one-dimensional Gaussian process with 256 pixels. To generate a Gaussian process with multiple dimensions, `shape` should be a tuple containing the number of pixels for each axis.
- `distances`: This should be the distances between the individual pixels in whatever units your code is using. For this demo we will set distances to 1.
- `**args`: The dictionary `args` contains the arguments parametrizing the model for the amplitude spectrum of the Gaussian process. We will discuss these arguments in detail below.
- As a final step, `cfm.finalize()` is executed, returning the final Gaussian process model.

The arguments described above are only a subset of the features of the `CorrelatedFieldMaker`. For additional options please see the API reference.

```
def fieldmaker(shape, distances, prefix, **args):
    cfm = jft.CorrelatedFieldMaker(prefix=f"{prefix}")
    cfm.set_amplitude_total_offset(
        offset_mean=args["offset_mean"], offset_std=args["offset_std"]
    )
    args.pop("offset_mean")
    args.pop("offset_std")
    cfm.add_fluctuations(
        shape=shape,
        distances=distances,
        **args,
    )
    cf_model = cfm.finalize()

    return cf_model, cfm.power_spectrum
```

5.2.2 `CorrelatedFieldMaker` parameters

The overview of the `CorrelatedFieldMaker` above left out detailed explanations about the `offset_mean` and `offset_std` arguments of the `set_amplitude_total_offset` method as well as the additional keyword arguments of the `add_fluctuations` method (which are `fluctuations`, `loglogavgslope`, `flexibility`, and `asperity`). These explanations should follow here.

The arguments `offset_mean` and `offset_std` passed to the `set_amplitude_total_offset` method control the prior distribution for the average value of the Gaussian process.

- `offset_mean` should be a float and is the mean value of the prior distribution on the average of the Gaussian process. Thus in the example of this *previous notebook* where we want to model the temperature in a room as a function of time with a Gaussian process and believe that the average temperature is 21 degrees, we would set `offset_mean = 21`.
- In many applications we don't know a priori the exact average value of the Gaussian process. In our example the actual average temperature might also be 22.7 degrees. With the argument `offset_std` we can specify how much we a priori believe that the actual mean might deviate from `offset_mean`. Thereby `offset_std` should be a tuple of two positive floats. With the first entry of the tuple we specify the standard deviation of the

prior for the average value. Thus in our example if we believe that the average temperature is 21 degrees, but think that the actual average temperature might be 21 ± 1 degrees, we would set the first entry in the tuple for `offset_std` to 1.0. The second entry in the tuple for `offset_std` specifies an uncertainty on the value set for the standard deviation (thus the first element). For many applications this functionality is not necessary and one can just insert a small positive float for the second entry in the tuple. In our example we could set `offset_mean = 21.` and `offset_std=(1, 1e-3)`, meaning that we effectively impose a Gaussian prior on the average temperature with mean 21 and standard deviation 1. If we are not sure about what standard deviation we want to impose we could set something larger than 10^{-3} in the second entry of the `offset_std` tuple which will be used as an uncertainty of the standard deviation. However, in most applications of the correlated field model this functionality is not needed.

The arguments `fluctuations`, `loglogavgslope`, `flexibility`, and `asperity` steer the prior model for the amplitude spectrum of the Gaussian process. Similar to `offset_std`, all of these arguments need to be tuples of floats. In the following we will explain the meaning of each of these parameters.

- The `fluctuations` parameter sets the standard deviation of the Gaussian process around its average value. The first entry in the `fluctuations` tuple is the mean value for the strength of the fluctuations and the second entry the uncertainty on it. In our example if we would estimate that the temperature fluctuates over time probably by 2 degrees, but are not sure and could imagine that the actual fluctuations might also be only by 1 degree or even by 3 degrees, we could set `fluctuation = (2., 1.)`.
- The `loglogavgslope` parameter sets the prior on the slope of the power law modeling the amplitude spectrum. The first entry in the tuple sets the prior mean on the slope of the power law, the second entry the uncertainty.
- The parameter `flexibility` sets the prior on how much the amplitude spectrum can deviate from a pure power law. Thereby the deviations from the power law are smooth functions themselves. As for the other parameters, the `flexibility` needs to be a tuple with the first entry being the prior mean of the strength of the deviations from a power law and the second entry being the standard deviation. If you don't want to allow for deviations from a pure power law amplitude spectrum, this part of the model can be disabled by setting `flexibility = None`.
- The `asperity` parameter is similar to `flexibility`. The difference to `flexibility` is that `asperity` models small scale deviations from a power law. For many applications `asperity` can be disabled by setting it to `None`, as small scale features in the amplitude spectrum correspond to oscillatory patterns (with a precise length scale) in the position space. However, for applications where oscillatory patterns are expected, as for example day/night temperature fluctuations, enabling `asperity` makes sense.

To visualize the Gaussian process model we initialize its parameters. We set the prior mean of the average value to 21 with a standard deviation of 1. Furthermore, we set the mean on the fluctuations of the Gaussian process to 2 with a standard deviation of 1. We set the average slope of the amplitude spectrum to -2 (with a std of 0.3) and activate the flexibility and asperity components to model deviations from a power law.

```
cf_kwargs = {
    "offset_mean": 21.0,
    "offset_std": (1.0, 1e-3),
    "fluctuations": (2.0, 1.0),
    "loglogavgslope": (-2.0, 0.3),
    "flexibility": (2.0, 1.0),
    "asperity": (1.0, 0.5),
    "prefix": "",
}
```

Using this set of parameters and the helper function from above we initialize a Gaussian process model. `model` will be the Gaussian process model itself and `ps` the power spectrum model.

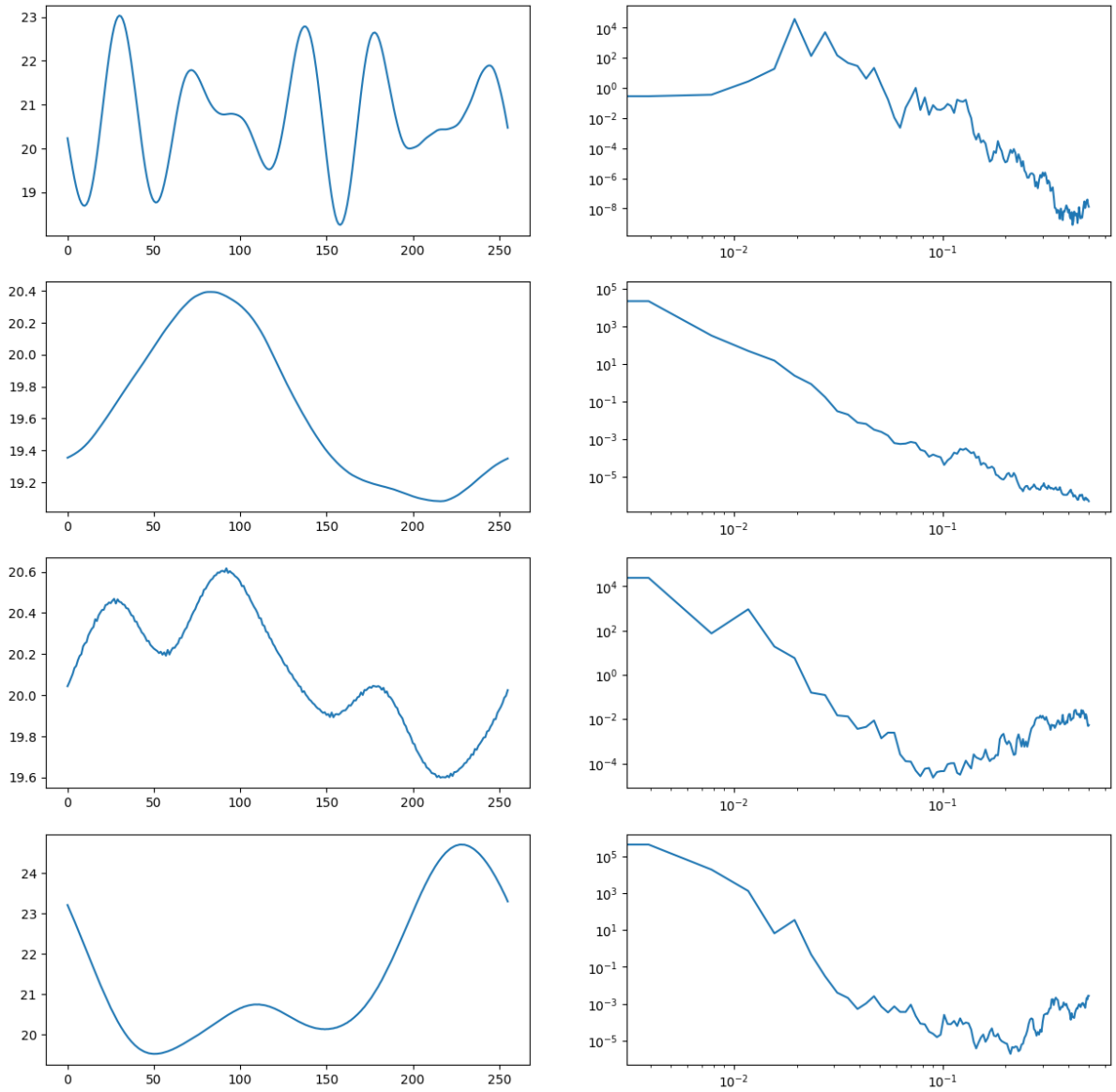
```
model, ps = fieldmaker(shape=shape, distances=distances, **cf_kwargs)
```

Next we want to look at random samples from this generative Gaussian process model. To do so we first initialize latent space standard normal samples on the input domain of the Gaussian process model.

```
key, *subkeys = random.split(key, 5)
xi_samples = [jft.random_like(sk, model.domain) for sk in subkeys]
```

Now we can plot for each latent space sample the corresponding Gaussian process sample and amplitude spectrum sample. In the left column are the Gaussian process samples and in the right the corresponding amplitude spectra. Note that the model implies periodic boundary conditions.

```
x_vec = model.target_grids[0].distances[0] * np.arange(
    shape
)  # sampling points of Gaussian process
k_vec = model.target_grids[
    0
].harmonic_grid.mode_lengths  # fourier modes of amplitude spectrum

fig, axes = plt.subplots(len(xi_samples), 2, figsize=(15, 15))
for i, xi in enumerate(xi_samples):
    axes[i, 0].plot(x_vec, model(xi))
    axes[i, 1].loglog(k_vec, ps(xi))
plt.show()
```



5.2.3 Parameter visualization

The remaining part of the notebook visualizes prior samples of the Gaussian process model to give an intuitive understanding of how to set the parameters. To do so, we define below a base choice of parameters and will then vary the parameters one by one to visualize the impact of each of them.

```
cf_kwargs = {
    "offset_mean": 0.0,
    "offset_std": (0.5, 0.2),
    "fluctuations": (1.0, 0.2),
    "loglogavgslope": (-2.0, 0.3),
    "flexibility": None,
    "asperity": None,
    "prefix": "",
}
```

Now, with a model generator `fieldmaker` and the grid in place, let us look at how varying each of these parameters changes the output power spectrum. For this, we build a function `vary_parameter` that takes a hyperparameter of the correlated field model, an array of values it takes, and outputs plots of the power spectra as given by the parameter it inputs.

```
realisations = 5


def vary_parameter(parameter, values, **args):
    global key
    for i, j in enumerate(values):
        syn_data = np.zeros(shape=(shape, realisations))
        syn_pow = np.zeros(shape=(int(shape / 2 + 1), realisations))
        args[parameter] = j
        fig = plt.figure(tight_layout=True, figsize=(10, 3))
        fig.suptitle(f"{parameter} = {j}")
        ax1 = fig.add_subplot(1, 2, 1)
        ax1.set_title("Field Realizations")
        ax1.set_ylim(
            -4.0,
            4,
        )
        ax2 = fig.add_subplot(1, 2, 2)
        ax2.set_xscale("log")
        ax2.set_yscale("log")
        ax2.set_title("Power Spectra")
        # Plotting different realisations for each field.
        for k in range(realisations):
            cf_model, pow_cf = fieldmaker(shape, distances, **args)
            key, signalk = jax.random.split(key, num=2)
            syn_signal = jft.random_like(signalk, cf_model.domain)
            syn_data[:, k] = cf_model(syn_signal)
            syn_pow[:, k] = pow_cf(syn_signal)
            ax1.plot(x_vec, syn_data[:, k], linewidth=1)
            ax2.plot(
                k_vec,
                np.sqrt(syn_pow[:, k]),
                linewidth=1,
            )
```

5.2.4 `loglogavgslope`

First, let us look at a hyperparameter that governs the power law nature of the power spectrum. Here, it is input as a tuple of (mean, std).

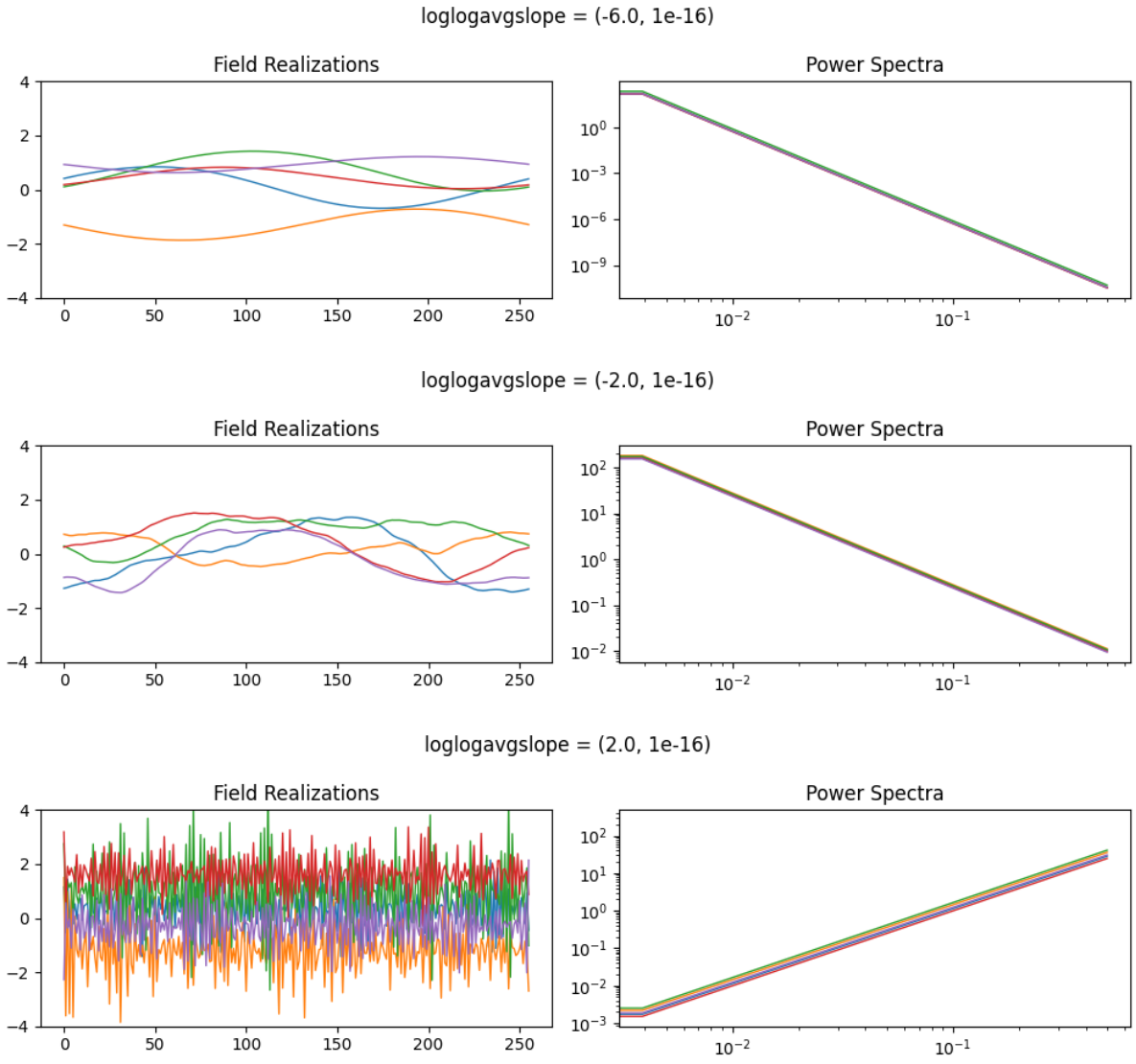
The `loglogavgslope` determines the steepness of the slope of the power spectrum in double logarithmic coordinates. In signal space, this is equivalent to determining the *smoothness* of the signal.

The `loglogavgslope` m is modeled with a Gaussian distribution, as it can take both negative and positive values. A negative slope indicates a falling power spectrum, and a positive slope indicates a rising power spectrum. The mean and standard deviation of the Gaussian prior are parameterized by μ_m and σ_m.

Let us look at how varying μ_m and σ_m affects the signal and power spectrum.

First, we vary μ_m, while keeping σ_m a low constant value. This denotes that we have very high confidence in the a priori value set for μ_m, but such a low value is used here only for demonstration purposes and does not reflect a practical choice of σ_m.

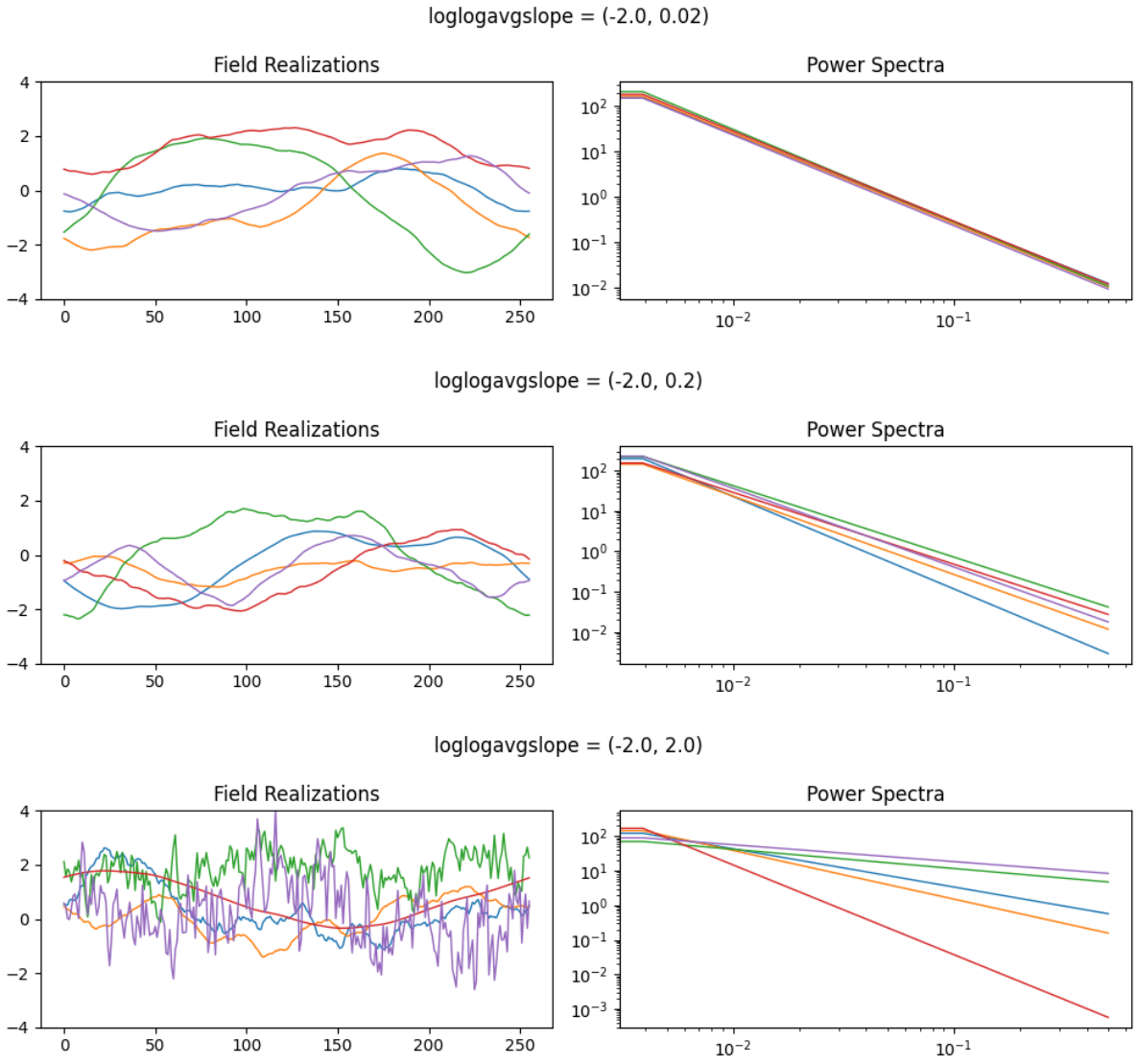
```
vary_parameter(
    "loglogavgslope",
    [(-6.0, 1e-16), (-2.0, 1e-16), (2.0, 1e-16)],
    **cf_kwargs,
)
```



We notice here that the steeper the falling slope, the smoother the signal realisations. This is intuitively understood as, the higher the power to the lowest modes (i.e. the largest scales) the more larger scale structures are present in the signal, which lends to the signal appearing smooth.

Now let us look at the influence of the variation of σ_m on the signal realisations and the power spectra.

```
vary_parameter("loglogavgslope", [(-2.0, 0.02), (-2.0, 0.2), (-2.0, 2.0)], **cf_
↪kwargs)
cf_kwargs["loglogavgslope"] = (-2.0, 1e-16)
```



Here we vary the relative absolute deviation of the `loglogavgslope` by 1%, 10% and 100%. This translates in the first case to the signal realisations being similarly smooth, in the second case to the signals varying in smoothness, and in the third case to the signals varying highly in smoothness. A higher value of the standard deviation of the `loglogavgslope` reflects our choice of being uncertain of the smoothness of the signal (or equivalently the steepness of the power spectrum).

5.2.5 `offset_mean`

The `offset_mean` parameter sets the average mean of the standardized signal $\bar{s}$. It acts in signal space, and hence does not have an effect on the power spectrum.

To inspect the variation of the following hyperparameters, we set a few of the hyperparameters to values that best show the difference in the signal and amplitude spectrum realizations: Here, `loglogavgslope` is set to a value of -3. Comparing to the plots above, we see that a value of -3 results in a relatively smooth signal, and a falling power spectrum.

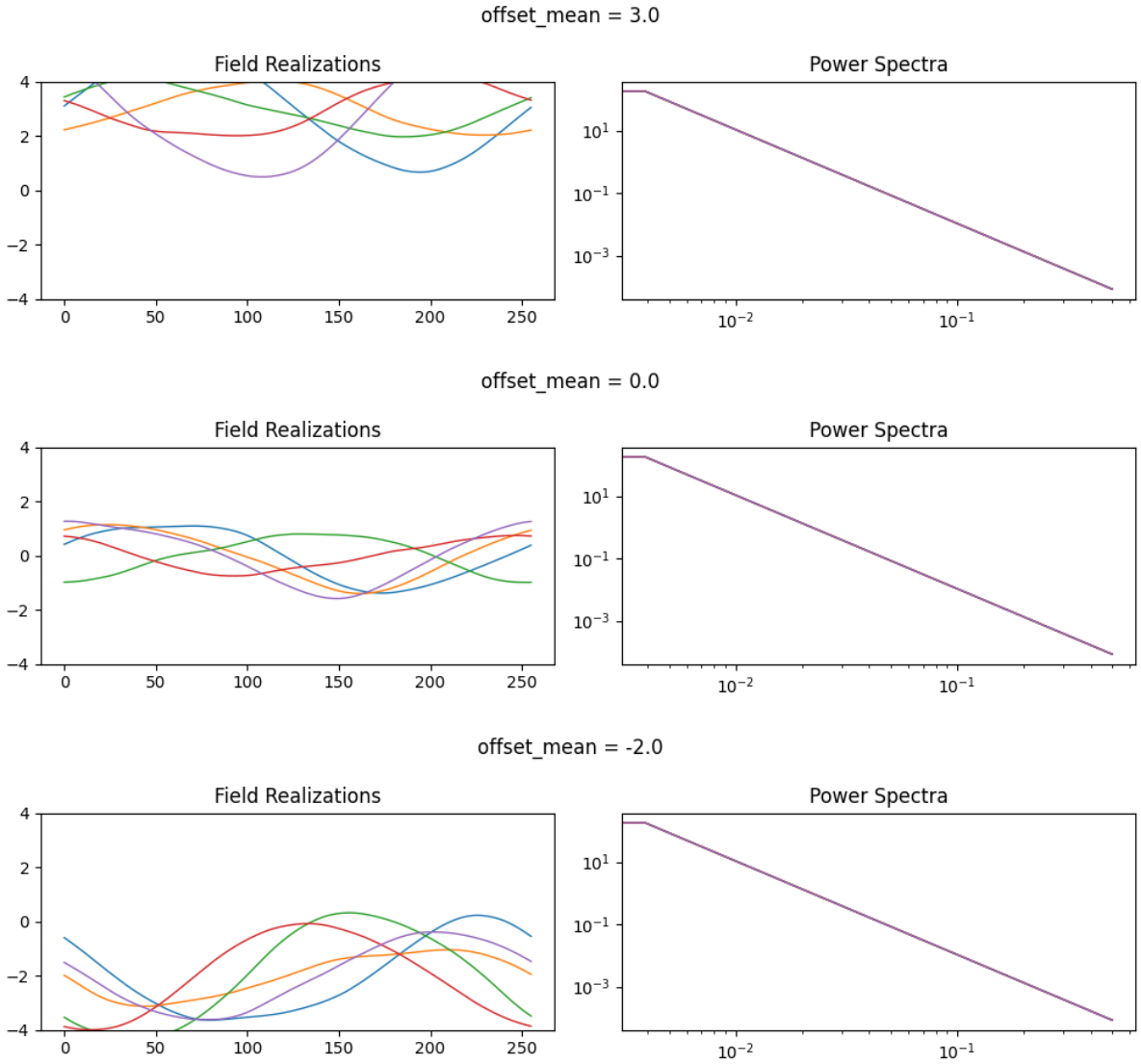
```
cf_kwargs["fluctuations"] = (1.0, 1e-16)
cf_kwargs["flexibility"] = (1e-3, 1e-16)
cf_kwargs["asperity"] = (1e-3, 1e-16)
```

(continues on next page)

(continued from previous page)

```
cf_kwargs["loglogavgslope"] = (-3, 1e-16)
cf_kwargs["offset_std"] = (1e-3, 1e-16)
```

```
vary_parameter("offset_mean", [3.0, 0.0, -2.0], **cf_kwargs)
cf_kwargs["offset_mean"] = 0.0
```



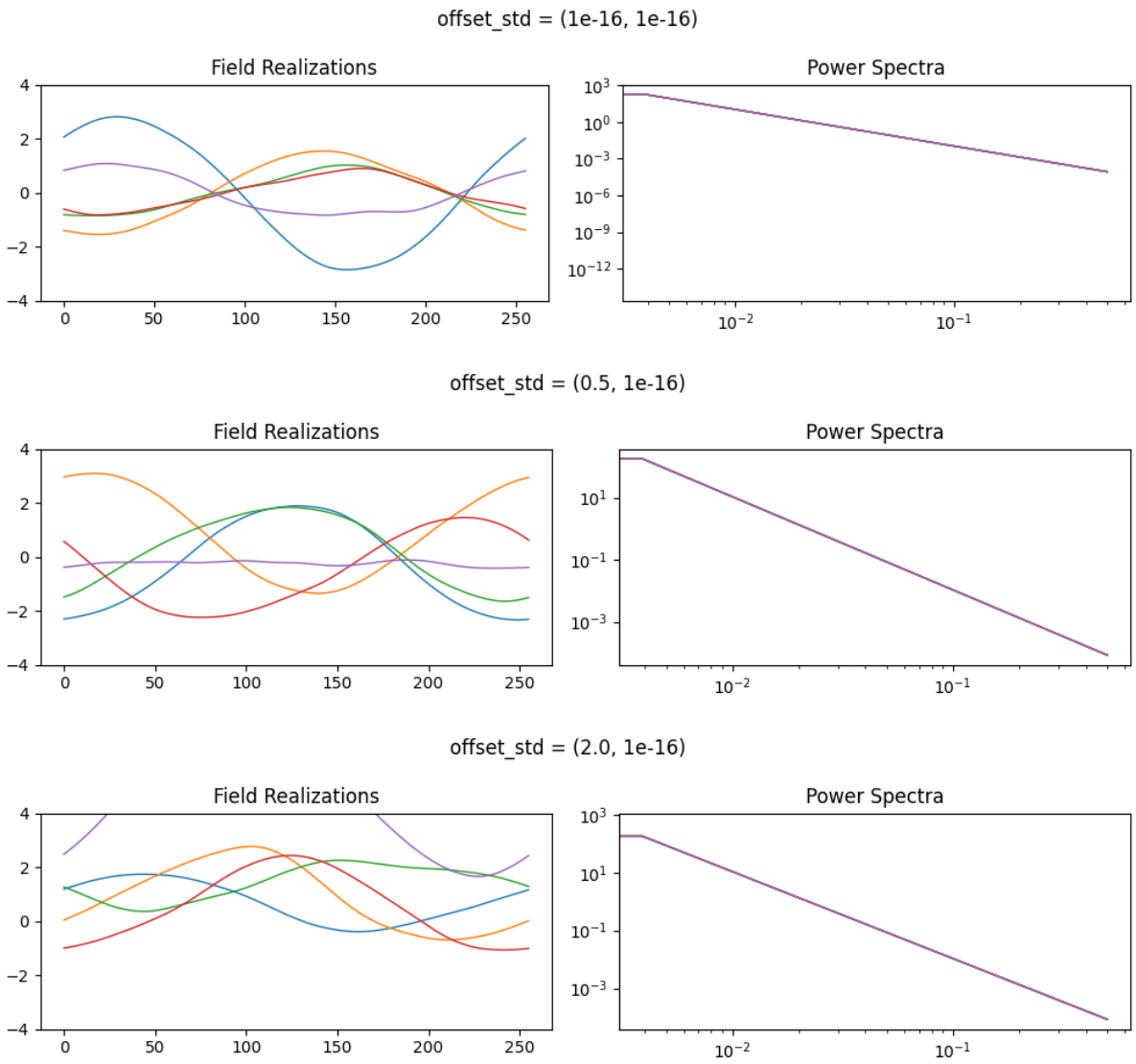
5.2.6 `offset_std`

The `offset_std` sets uncertanty of the average value of the signal. Although we model the signal here in one dimension, in case the signal has multiple dimensions, `offset_std` determines the global zero-mode of all subdomains. It is input as a tuple of (mean, std).

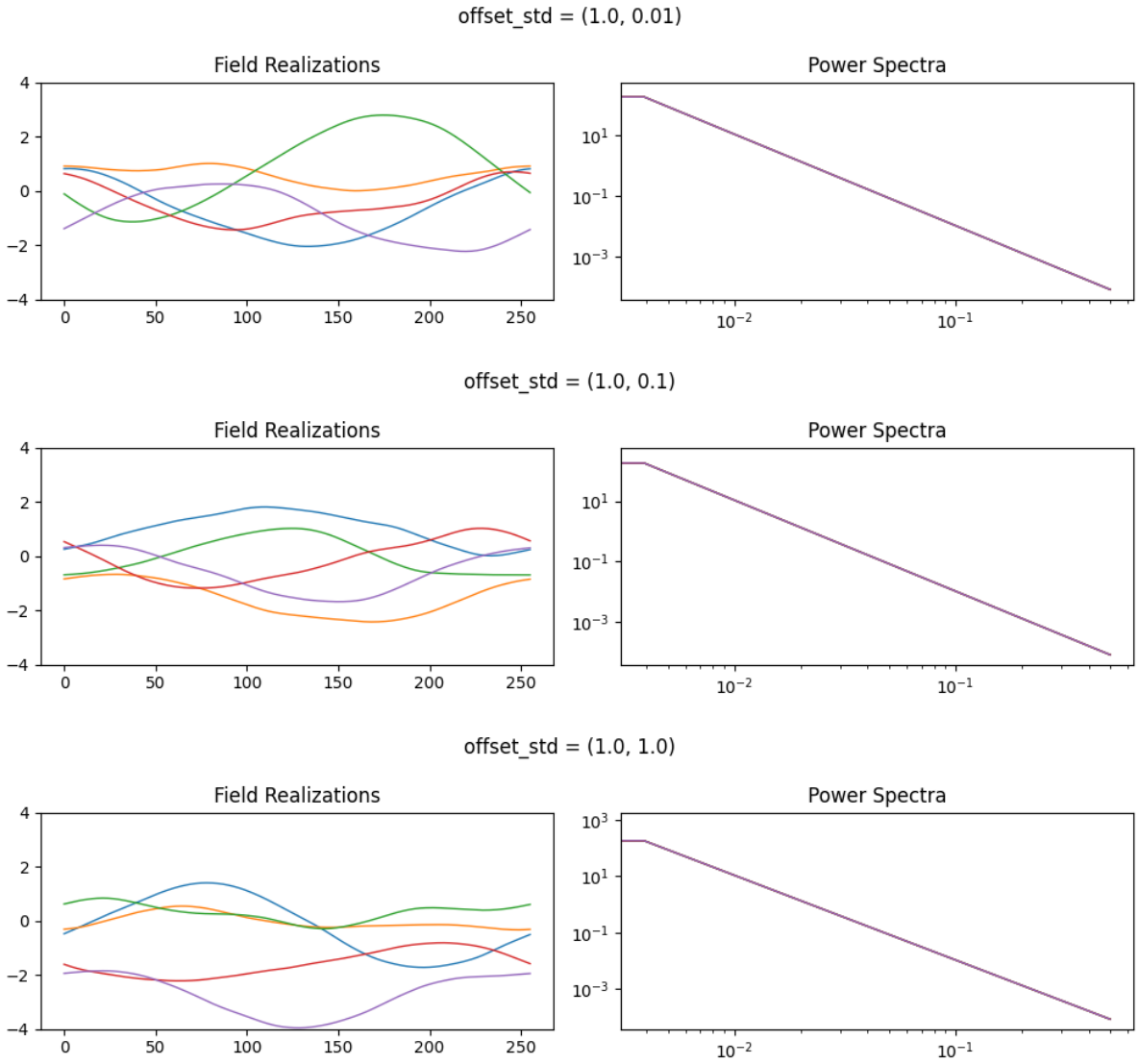
`offset_std` must always be positive, and hence is modeled with a lognormal prior. The lognormal prior is parametrized by setting the mean μ_α and the standard deviation σ_α of the lognormal distribution.

We start by inspecting variations in μ_α, and then σ_α.

```
vary_parameter("offset_std", [(1e-16, 1e-16), (0.5, 1e-16), (2.0, 1e-16)], **cf_
↪kwargs)
```




```
vary_parameter("offset_std", [(1.0, 0.01), (1.0, 0.1), (1.0, 1.0)], **cf_kwargs)

cf_kwargs["offset_std"] = (0.5, 0.2)
```

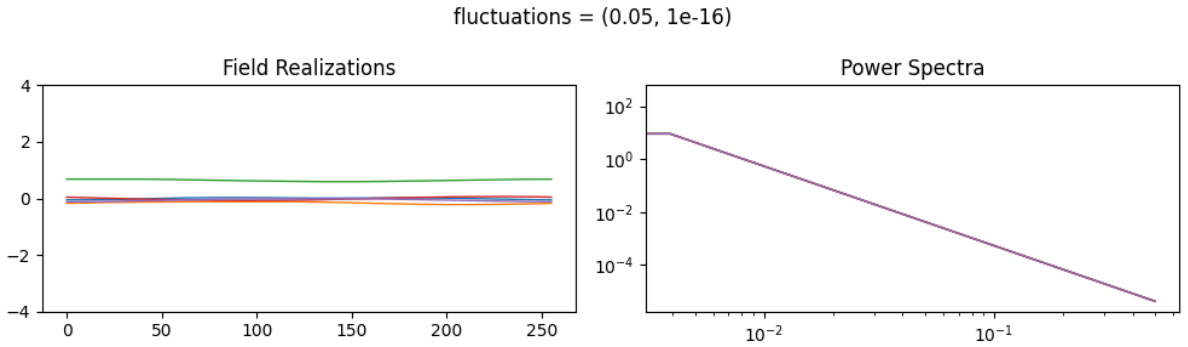


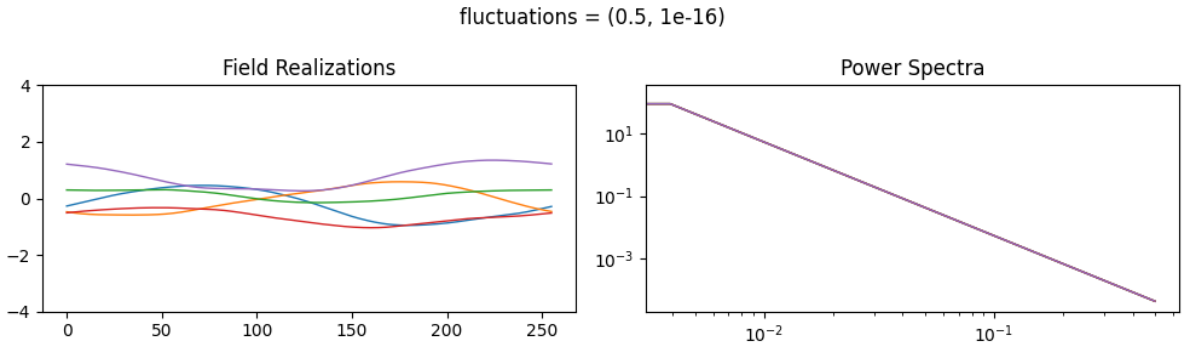
5.2.7 `fluctuations`

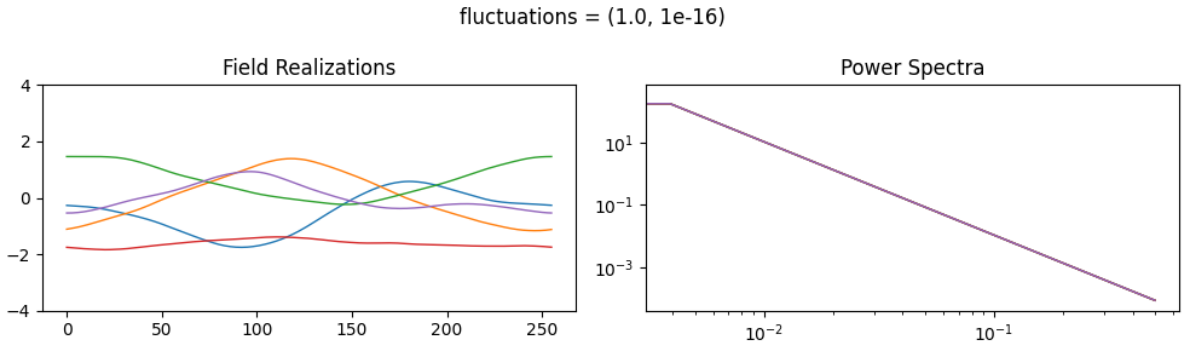
The `fluctuations` parameter steers how much the samples fluctuate around the average value. In other words, with the `fluctuations` parameter we set the prior distribution on the standard deviation of the Gaussian process. As the standard deviation needs to be positive we model it with a lognormal distribution. As before we parametrize the lognormal distribution by it's mean and standard deviation.

Again, let us first look at the influence of μ_a on the signal realisations, then at the one of σ_a.

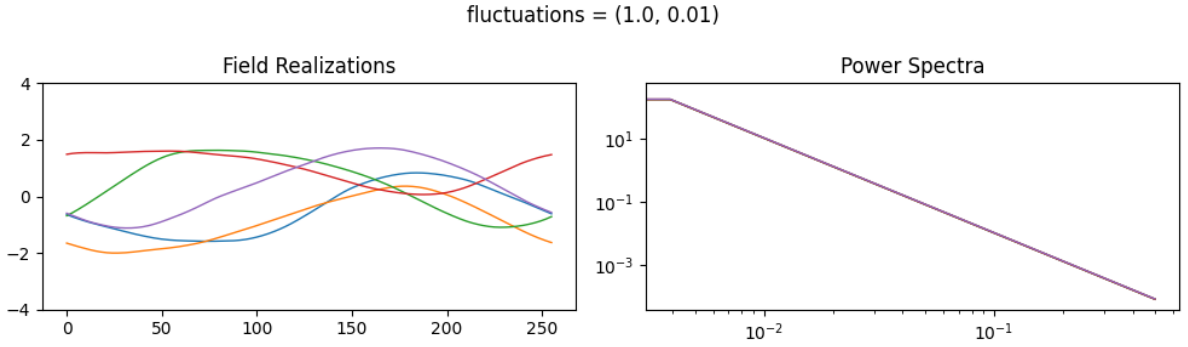
```
vary_parameter("fluctuations", [(0.05, 1e-16), (0.5, 1e-16), (1.0, 1e-16)], **cf_
↪kwargs)
cf_kwargs["fluctuations"] = (1.0, 1e-16)
```

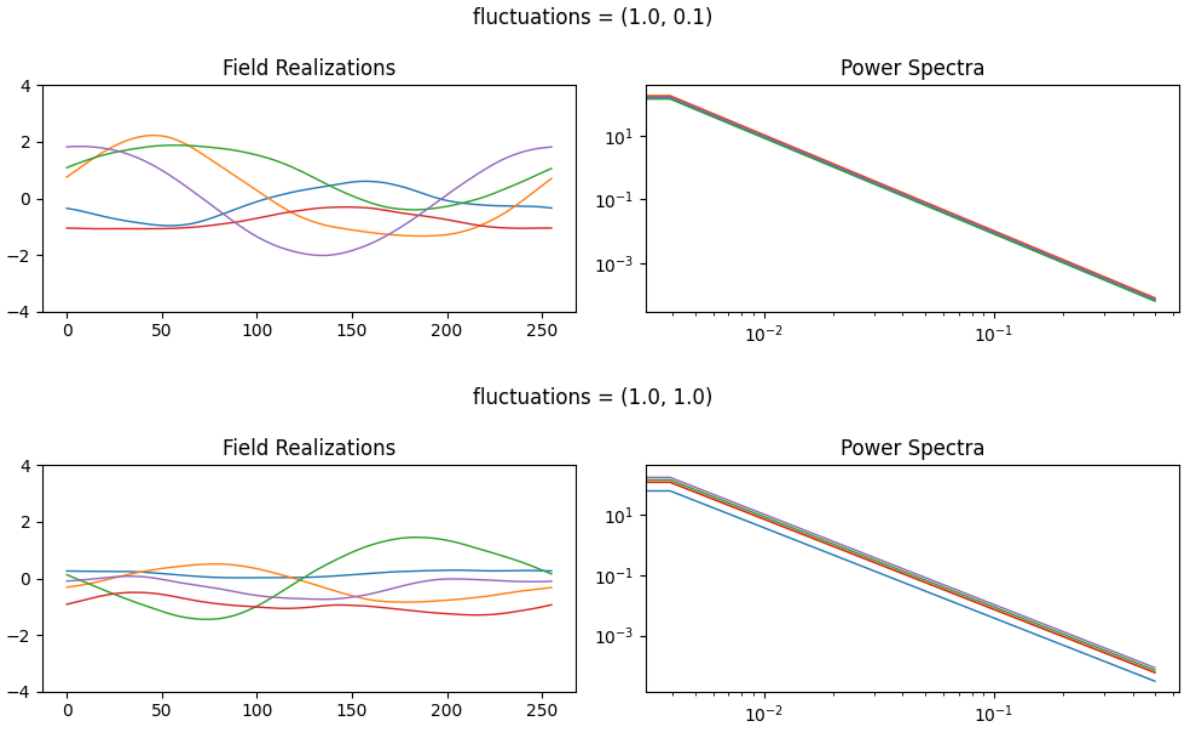







```
vary_parameter("fluctuations", [(1.0, 0.01), (1.0, 0.1), (1.0, 1.0)], **cf_kwargs)
cf_kwargs["fluctuations"] = (1.0, 1e-16)
```





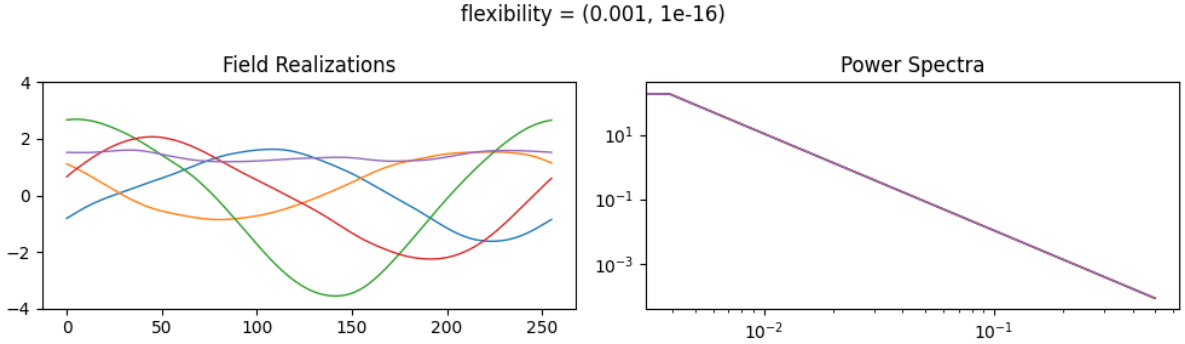
5.2.8 `flexibility` and `asperity`

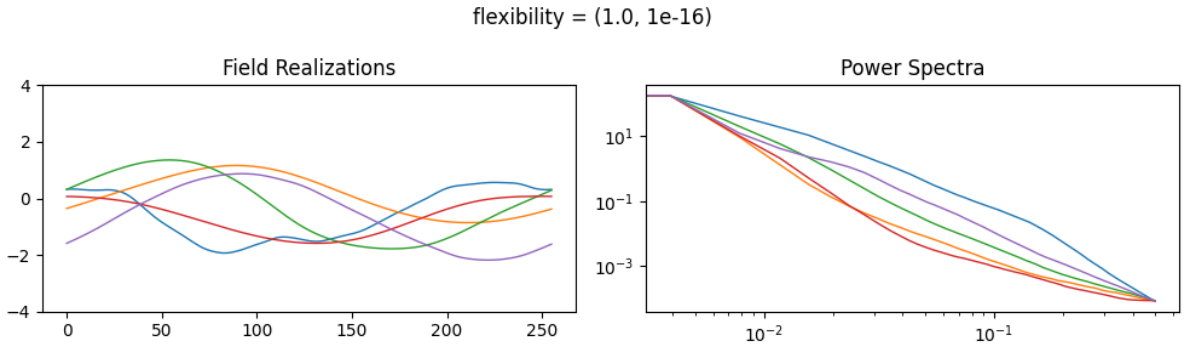
There are two hyperparameters remaining: `flexibility` and `asperity`. The flexibility η sets the amplitude of the deviations of the power spectrum from a power law on the double logarithmic scale, and the asperity ϵ sets the roughness of the deviations of the power spectrum. Internally in the correlated field model these deviations are modeled with a Wiener process and an integrated Wiener process. `flexibility` corresponds to the amplitude of the Integrated Wiener Process (IWP) component of the power spectrum, and `asperity` corresponds to the roughness of the IWP component.

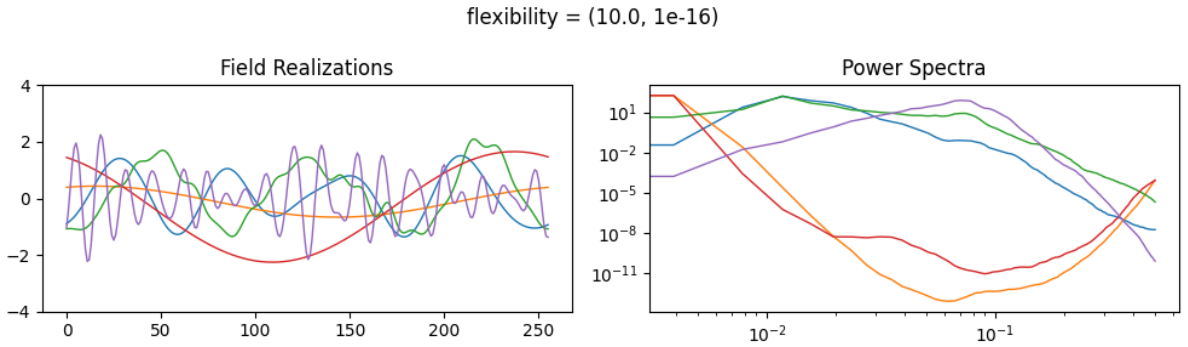
Both `asperity` and `flexibility` are required to be positive, and are modeled with a lognormal prior, for which we can set the mean and variance.

First, let us look at the impact on the signal and power spectra upon varying `flexibility`. First, we vary μ_η and then σ_η.

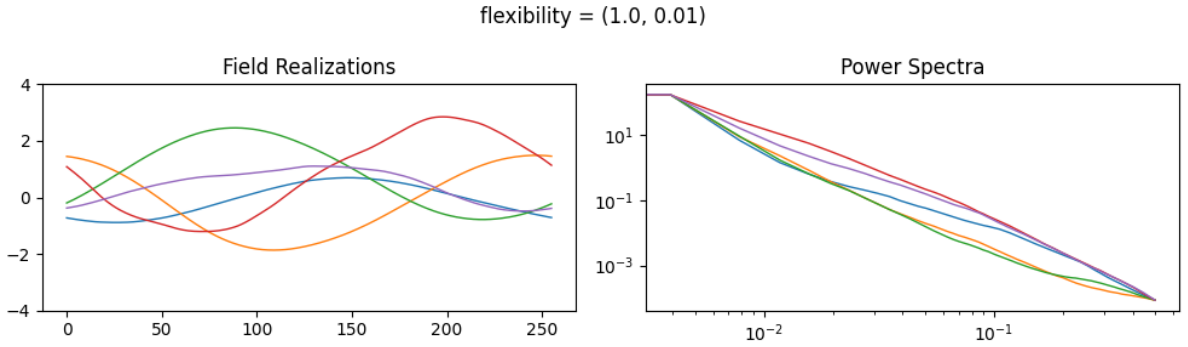
```
vary_parameter(
    "flexibility", [(0.001, 1e-16), (1.0, 1e-16), (10.0, 1e-16)], **cf_kwargs
)
```

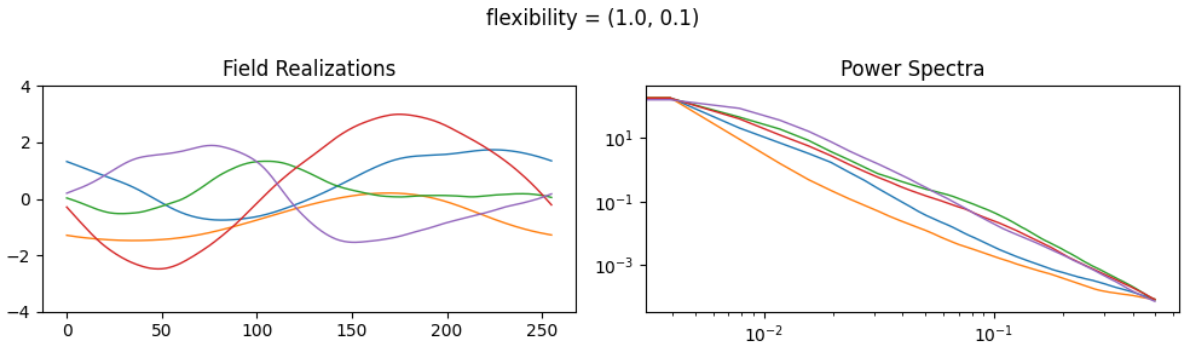


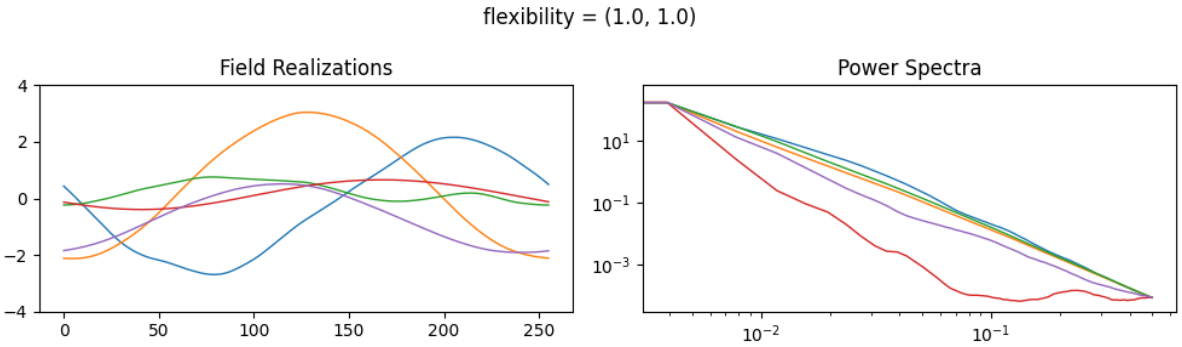





```
vary_parameter("flexibility", [(1.0, 0.01), (1.0, 0.1), (1.0, 1.0)], **cf_kwargs)
```



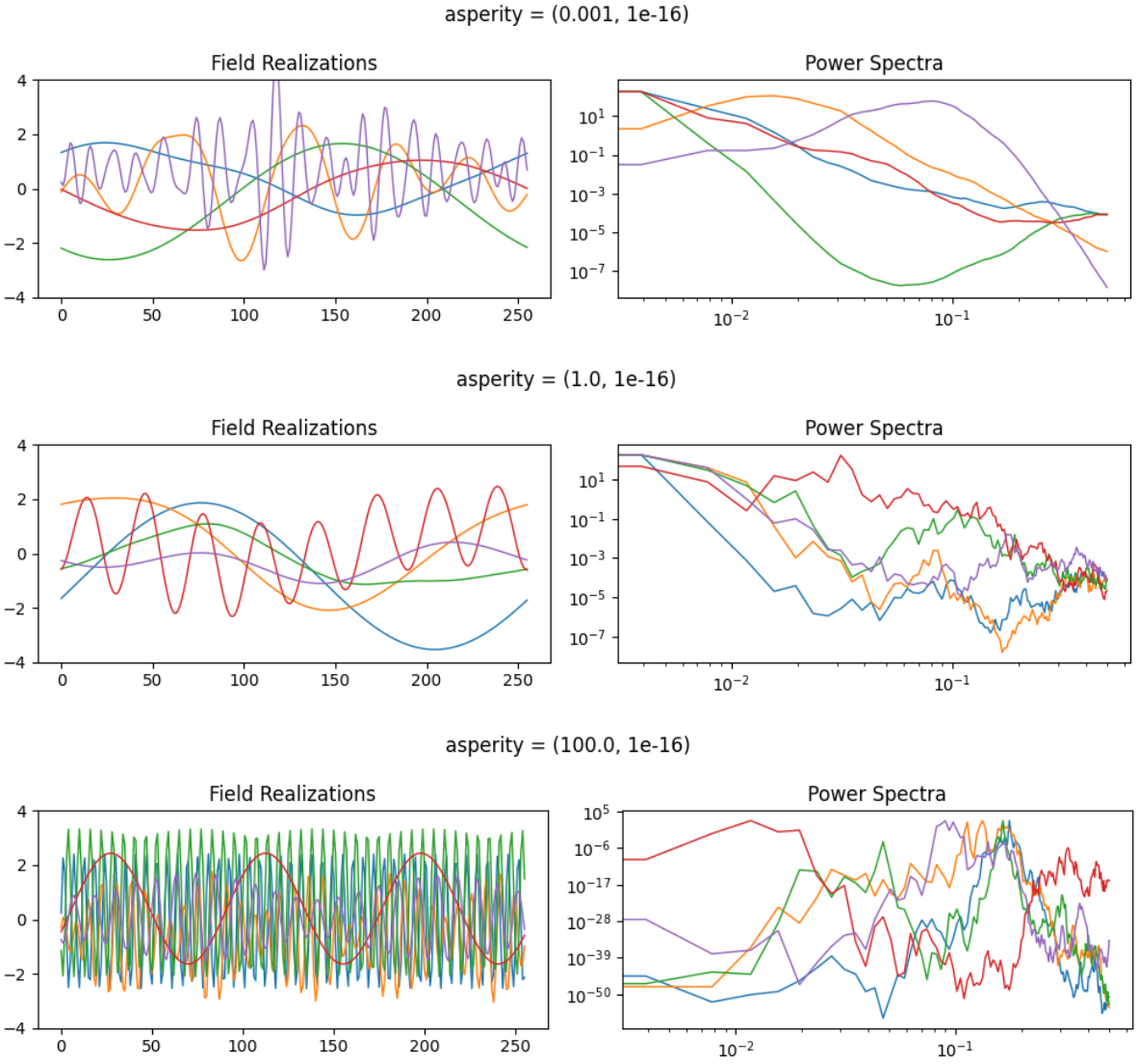




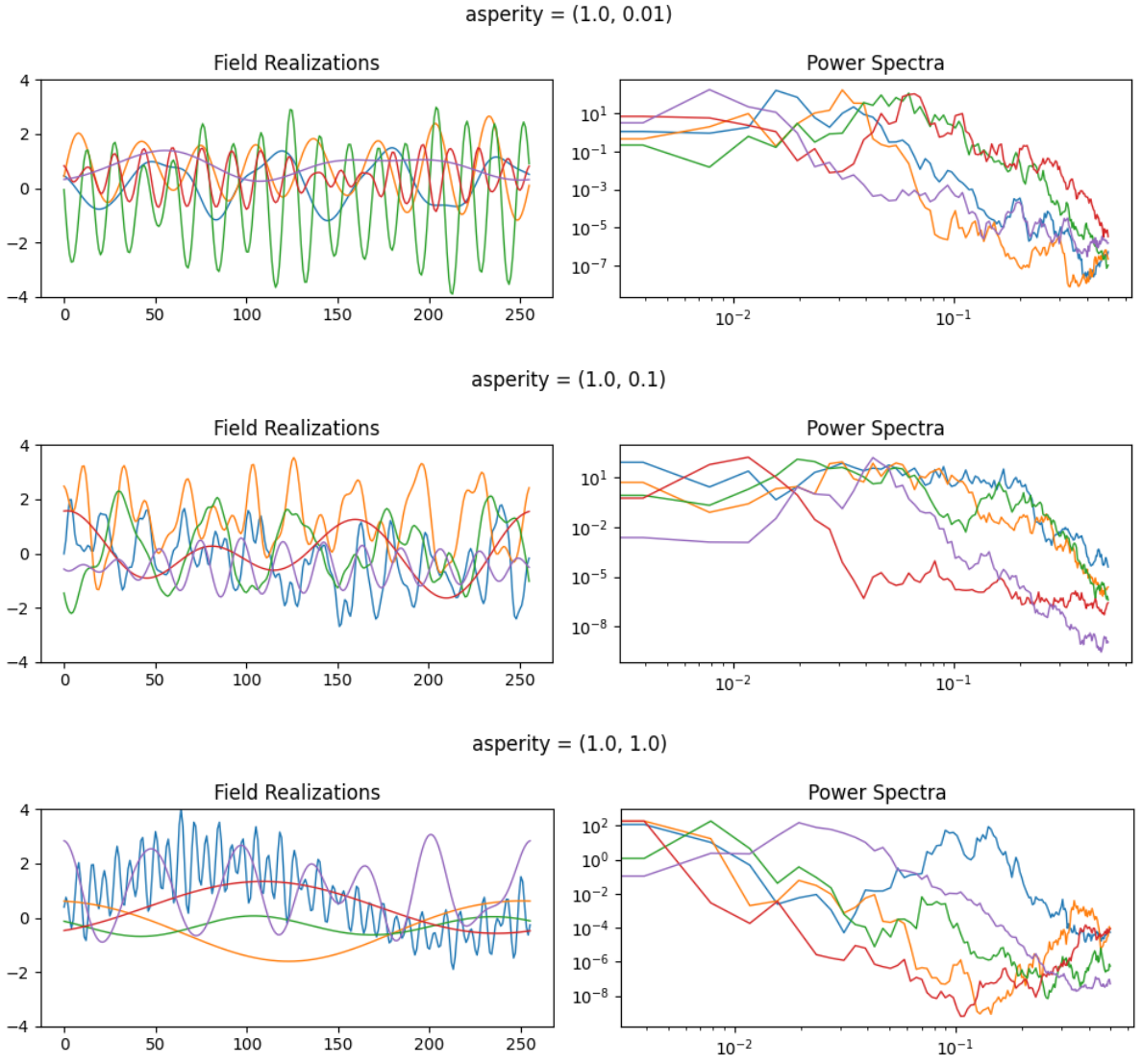
To look at `asperity`, we reset the value of `flexibility` to 5.0.

```
cf_kwargs["flexibility"] = (5.0, 1e-16)
```

```
vary_parameter("asperity", [(0.001, 1e-16), (1.0, 1e-16), (100.0, 1e-16)], **cf_
↪kwargs)
```



```
vary_parameter("asperity", [(1.0, 0.01), (1.0, 0.1), (1.0, 1.0)], **cf_kwargs)
cf_kwargs["asperity"] = (1.0, 1e-16)
```



5.3 Summary

In this notebook we discussed the correlated field model, where we set up a non-parametric generative model to infer the powerspetrum of the Gaussian process additionally to the process itself. The hyperparameters of the `CorrelatedFieldMaker` are the steepness of the slope `loglogavgslope`, the average mean of the signal `offset_mean` and the standard deviation of this average mean `offset_std`, the standard deviation of the Gaussian process `fluctuations`, as well as deviations from a power law `flexibility` and its roughness `asperity`.

CHAPTER

SIX

EVENT RATE RECONSTRUCTION

This notebook explains how to model a process that produces integer-valued count data, for example, from a photon source or the decay of a radioactive element. It introduces the Poisson distribution as the corresponding likelihood and the constraints that must be employed to model the rate function of the underlying Poisson process. If you are not familiar with NIFTy, please start with the notebooks *Models* and *Inference*.

```
import nifty.re as jft

import numpy as np
import jax.numpy as jnp
import jax.random as random

%matplotlib inline
import matplotlib.pyplot as plt

plt.rcParams["figure.dpi"] = 100

from functools import partial

seed = 42
rng_key = random.PRNGKey(seed)
```

6.1 Poisson process

A Poisson process is a counting process with a strictly positive rate function $\rho(x)$. Under a counting process, we can understand a stochastic process that keeps track of all registered events in a given space $\mathcal{S}$. The rate function $\rho(x)$ describes the rate of events in an infinitesimal volume in $\mathcal{S}$. Furthermore, for a Poisson process, the registered events in a region Ω are distributed by a Poisson distribution

$$P(n|\lambda) = \frac{\lambda^n \, e^{-\lambda}}{n!}$$

with $n \in \mathbb{N}_0$ being the number of registered events and

$$\lambda = \int_\Omega dy \, \rho(y)$$

being the expected counts in the region Ω. The corresponding Hamiltonian $\mathcal{H}(n|\lambda)$ to the Poisson distribution $P(n|\lambda)$ reads

$$\mathcal{H}(n|\lambda) = -\ln\left[P(n|\lambda)\right] = \lambda - n\ln(\lambda) + \ln(n!)$$

6.2 Log-normal Poisson model

The generative log-normal Poisson model first starts with a normally distributed random field $s(x)$ drawn from a normal distribution $\mathcal{G}(s, S)$. To ensure positivity and allow for variations over several orders of magnitude, $s(x)$ will not directly be used as a model for the rate function, but rather as the logarithm of the rate function scales. Consequently, the functional relation between $\rho(x)$ and s^x reads as follows

$$s(x) = \ln\left(\frac{\rho(x)}{\rho_0}\right) \iff \rho(x) = \rho_0\, \mathrm{e}^{s(x)}.$$

Moreover, ρ_0 is chosen such that it ensures $\langle s \rangle_{(s)} = 0$.

Similar to the definition of expected counts for the Poisson process, applying a response $R(x)$ to the rate function ρ^x gives the expected count

$$\lambda = \int_\Omega dx\, R(x)\, \rho(x)$$

in a region Ω. For Poisson count observations, R can, for example, contain exposure maps, point spread functions, or masking operations.

For a set of bins $\{\Omega_i\}_{i=1}^{N_{\text{bin}}}$, the log-normal Poisson model assumes a Poissonian $\mathcal{P}(d_i|\lambda_i)$ likelihood for every single bin $i \in (1, \dots, N_{\text{bin}})$. We can assume that these bins are different regions $\{\Omega_i\}_{i=1}^{N_{\text{bin}}}$ for which we can ask how probable it is that a certain number of events occurred. Assuming independent noise for each bin, the joint likelihood

$$\mathcal{P}(d|\lambda) = \prod_i \mathcal{P}(d_i|\lambda_i) = \prod_i \frac{(\lambda_i)^{d_i}\, e^{-\lambda_i}}{d_i!}$$

is just a product of all single bin likelihoods.

6.2.1 Log-normal Poisson model with the correlated field

This section should be an example of how to model Poisson count data in time. Naturally, we can extend the procedure to multiple dimensions, similar to this example.

First, let us read the data contained in `data_poisson.txt`.

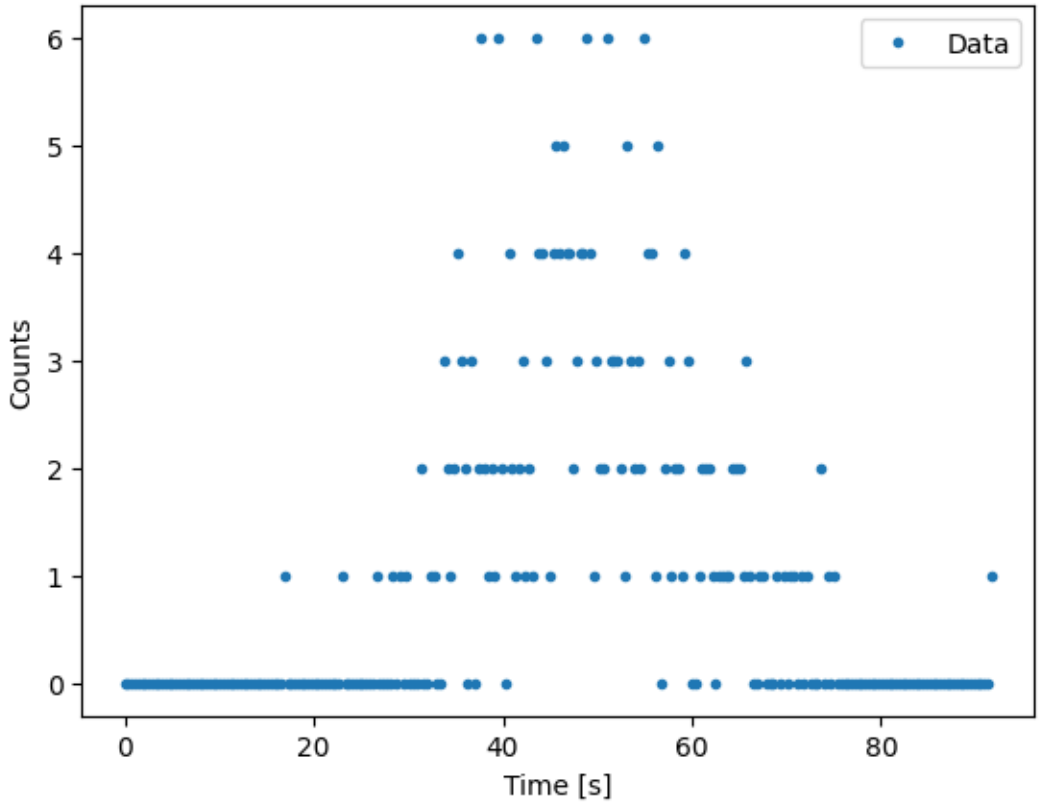
```
events = np.loadtxt("data_poisson.txt")

for k in range(15):
    print(events[k])
```

```
17.025680541992188
23.017667770385742
26.773649215698242
28.523881912231445
29.105871200561523
30.136960983276367
31.293071746826172
31.614225387573242
32.483299255371094
32.93983840942383
33.88088607788086
33.984371185302734
34.11470031738281
34.33364486694336
34.449947357177734
```

`events` now contains all recorded events' time stamps, but there is still no count data. Therefore, we take all events and bin them over the viewed interval $[0, T]$ with a resolution of `t_res`. In this analysis, we choose the last timestamp as our T. Using `jnp.histogram`, we get the counts per bin and its bin bounds. Moreover, by only setting the number of desired bins, we get uniform bins with bin width Δt.

Now let us visualize the binned count data.

```
t_res = 2**8
T = events.max()
data, bins = jnp.histogram(events, t_res, (0, T))
data = data.astype(int)
bins = bins[:-1]

plt.xlabel("Time [s]")
plt.ylabel("Counts")
plt.plot(bins, data, ".", label="Data", color="tab:blue")
plt.legend(loc="upper right")
plt.show()
```



The plot shows us that roughly between 30 and 70 seconds, we have more counts than at the beginning and end of the interval. Therefore, it is plausible that the underlying rate function $\rho(t)$ must be higher in this region, meaning $\rho(t)$ allows for a greater number of events.

Now let us model the rate function $\rho(t)$ itself.

The exact form of $\rho(t)$ is unavailable in a real-world scenario. Therefore, we model the rate function $\rho(t)$ via the log-normal Poisson model. We choose a correlated field with a non-parametric power spectrum for the underlying Gaussian process. Since the correlated field uses the Hartley transformation, it follows periodic boundary conditions. To reconstruct also non-periodic signals, we make use of zero padding. For that, we define our correlated field for twice the time interval and then cut out the zero-padded region in the forward model. Furthermore, we choose the `distances` of the correlated

field to coincide with the bin width of the data. This saves us transformations between (differently binned) data and Poisson rates, simplifying operators or fields of the model for the expected counts.

```
cf_offset_dct = dict(
    offset_mean=1,
    offset_std=(0.5, 0.1),
)

cf_ps_dct = dict(
    shape=2 * t_res,
    distances=T / t_res,
    non_parametric_kind="power",
    fluctuations=(0.5, 0.1),
    loglogavgslope=(-4, 0.5),
    flexibility=(1, 0.5),
    asperity=(10, 5),
)

cf_maker = jft.CorrelatedFieldMaker("signal_")
cf_maker.set_amplitude_total_offset(**cf_offset_dct)
cf_maker.add_fluctuations(**cf_ps_dct)
s = cf_maker.finalize()
```

The forward model mainly consists of the log-normal Poisson model described at the beginning.

1. We choose the correlated field's timeline to coincide with the data's timeline in the first half. The second half is the artificial zero padding region and must be cut out. The method `signal` in the forward model applies this slicing operation.
2. Next, we ensure positivity by exponentiating the resulting signal in the `exp_signal` method. By making it a separate method, we can easily extract $\rho(t)$ later if needed.
3. Lastly, we compute the expected counts from the Poisson rate. Recall that the log-normal Poisson model computes the expected counts per bin i within the time interval $[t_i, t_{i+1}]$ as

$$\lambda_i = \int_{t_i}^{t_{i+1}} d\tau \, R_i(\tau) \, e^{s(\tau)}.$$

In our case, the response operator R is the identity operator. As one entry of the signal field s corresponds to exactly one data bin, we can assume that $s(t) = s(t_i) = s_i$. The integral simplifies to

$$\lambda_i = e^{s_i} \int_{t_i}^{t_{i+1}} d\tau \; = e^{s_i} \, (t_{i+1} - t_i) = e^{s_i} \, \Delta t.$$

Thus, the `__call__` method of the forward model takes the expontiated signal and multiplies all array entries by the bin width.

```
class LogNormalPoisson(jft.Model):
    def __init__(self, signal_field, res, T):
        self.s = signal_field
        self.res = res
        self.dt = T / res

        super().__init__(init=signal_field.init)

    def __call__(self, x):
        return self.dt * self.exp_signal(x)
```

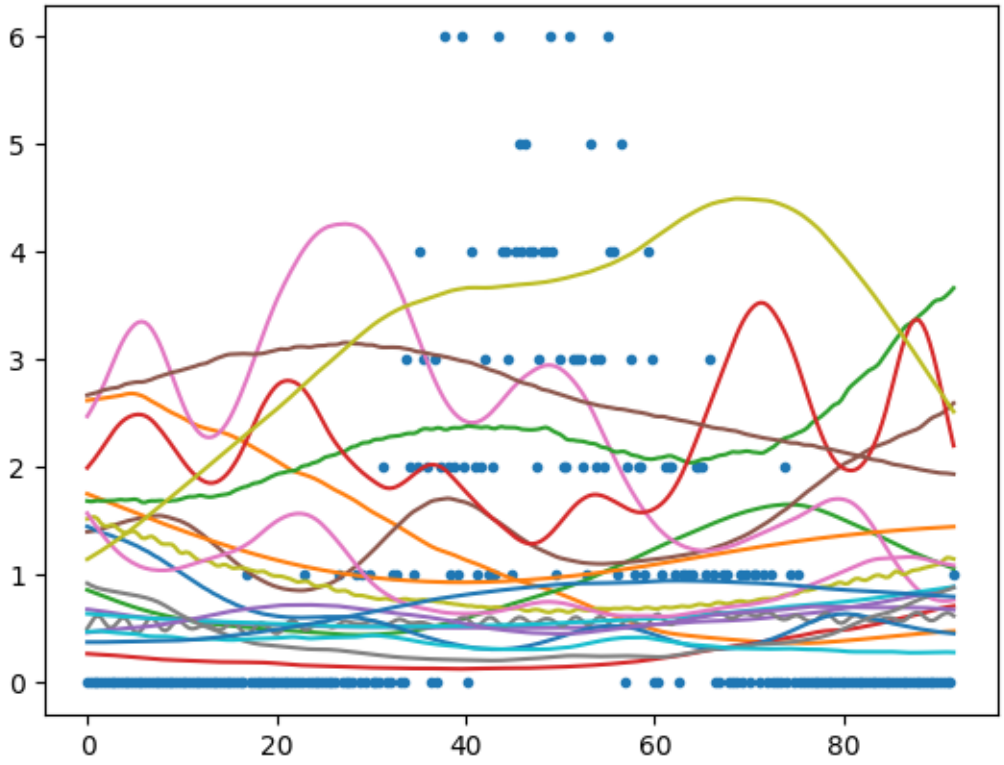
(continues on next page)

(continued from previous page)

```
    def exp_signal(self, x):
        return jnp.exp(self.signal(x))

    def signal(self, x):
        return self.s(x)[: self.res]


lamb = LogNormalPoisson(s, t_res, T)
```

To see if the chosen hyperparameters of our correlated field prior are reasonable, let us look at some prior samples of our forward model.

```
plt.plot(bins, data, ".")
key, k1 = random.split(rng_key, 2)
for k in range(20):
    k0, k1 = random.split(k1, 2)
    plt.plot(bins, lamb(lamb.init(k0)))
```



The plot above displays the data count by the points, and 20 prior samples as graphs. We can see that the hyperparameters are set to reasonable values, as the different prior samples show enough variance in amplitude and offset to cover almost every possible data point. Furthermore, the different realizations show that the variance on the hyperparameters allows for a wide range of possible realizations, thus enabling the search for the correct values of the hyperparameters.

Next, we have to define the likelihood of our inference problem. For count data, as we are using the log-normal Poisson model, we choose a Poissonian likelihood. `NIFTy.re` implements a Poissonian likelihood as the class `jft.Poissonian`. The class itself needs the binned count data and computes the Information Hamiltonian of a multivariate Poisson distribution

$$\mathcal{H}(d|\lambda) = \sum_i \mathcal{H}(d_i|\lambda_i) = \sum_i \lambda_i - d_i \ln(\lambda_i) + \ln(d_i!)$$

up to constant terms in λ_i.

As the likelihood does not know about the model, we must amend an instance of the model class.

```
lh = jft.Poissonian(data).amend(lamb)
```

The last part is again the inference algorithm. For an introduction on using the `optimize_kl` function, please look at the *inference notebook*. For visualization, we evaluate the mean and standard deviation of the exponentiated signal $e^{s(t)}$ after each VI iteration and plot the mean, together with a 1σ -band, against the exact rate function $\rho(t)$. Furthermore, the reconstructed power spectrum of the correlated field is shown next to it.

```
def callback(samples, state):
    exp_mean, exp_std = jft.mean_and_std(tuple(lamb.exp_signal(s) for s in samples))
    ps_mean, ps_std = jft.mean_and_std(
        tuple(cf_maker.power_spectrum(s) for s in samples)
    )

    fig, axs = plt.subplots(nrows=1, ncols=2, figsize=(10, 5), constrained_
 ↪layout=True)

    axs[0].plot(bins, exp_mean, label="Posterior mean", color="tab:blue")
    axs[0].fill_between(
        bins, exp_mean - exp_std, exp_mean + exp_std, color="tab:blue", alpha=0.3
    )
    axs[0].legend()

    axs[1].set_xscale("log")
    axs[1].set_yscale("log")
    axs[1].plot(
        s.target_grids[0].harmonic_grid.mode_lengths,
        ps_mean,
        label="Posterior power spectrum",
        color="tab:blue",
    )

    plt.show()
```

We will now use the geometric Variational Inference (geoVI) algorithm that is better in approximating non-Gaussian posteriors than Metric Gaussian Variational Inference (MGVI), as the posterior will generally be non-Gaussian. Let's motivate this by looking at the joint distribution of data d and the expectation value λ in one bin. Considering only one bin, we have a one-dimensional Gaussian and Poisson distribution for the prior and likelihood. Additionally, we assume a unit response in the log-normal model, meaning that $\lambda(s) = e^s$.

$$\mathcal{P}(s) = \mathcal{G}(s, \sigma^2) = \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{s^2}{2\sigma^2}}$$
$$\mathcal{P}(d|\lambda(s)) = \frac{\lambda^d e^{-\lambda}}{d!}$$

Since λ is defined by an exponentiated normally distributed s, it will follow the log-normal distribution

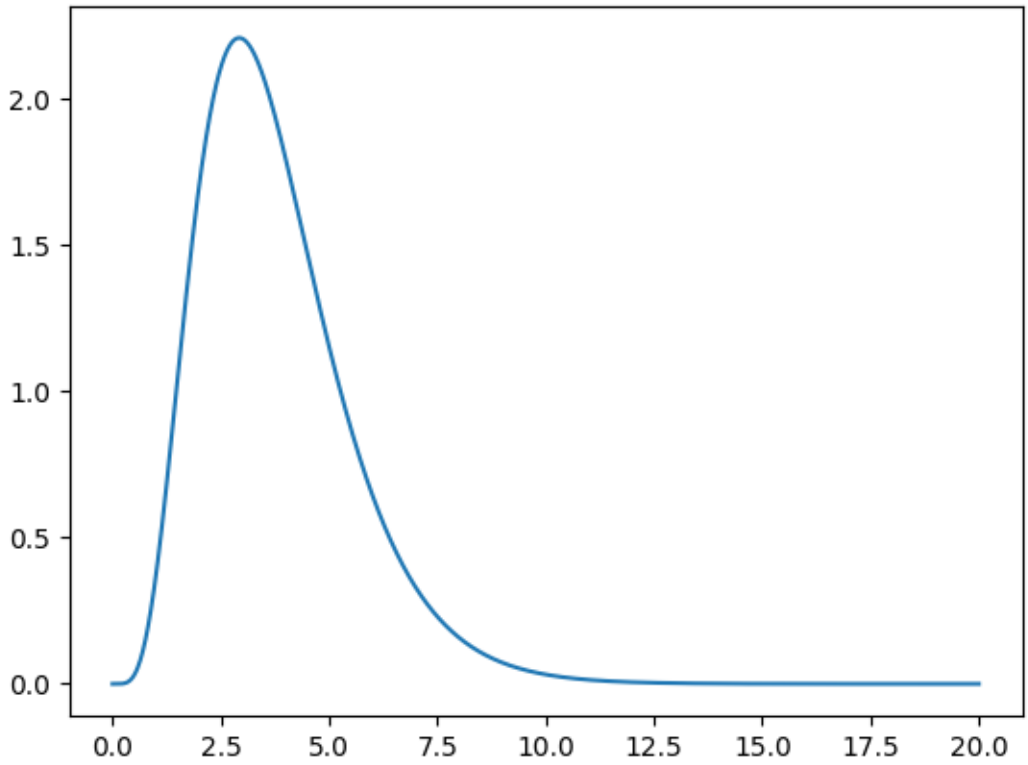
$$\mathcal{P}(\lambda) = \frac{1}{\sqrt{2\pi}\sigma\,\lambda} \exp\left[-\frac{\ln^2(\lambda)}{2\sigma^2}\right].$$

The joint distribution of d and λ will therefore reads

$$\mathcal{P}(d, \lambda) = \mathcal{P}(d|\lambda)\,\mathcal{P}(\lambda) \propto \lambda^{d-1} \exp\left[-\lambda - \frac{\ln^2(\lambda)}{2\sigma^2}\right].$$

Now, let us visualize the unnormalized posterior distribution.

```
def joint_PDF_curve(x, d, sigma):
    return x ** (d - 1) * jnp.exp(-x - jnp.log(x) ** 2 / (2 * sigma**2))


x = jnp.linspace(0, 20, 1000)
y = joint_PDF_curve(x, 5, 1)

plt.plot(x, y)
plt.show()
```



As we can see, the posterior

$$\mathcal{P}(\lambda|d) = \frac{\mathcal{P}(d,\lambda)}{\mathcal{P}(d)} \propto \lambda^{d-1} \exp\left[-\lambda - \frac{\ln^2(\lambda)}{2\sigma^2}\right]$$

deviates from a Gaussian posterior. Therefore, using MGVI in the inference scheme may lead to a suboptimal reconstruction. Now, let us use the geoVI (see https://arxiv.org/abs/2105.10470) algorithm, which uses the Fisher Information metric to construct non-Gaussian posterior distributions.

Now let us specify the `optimize_kl` arguments.

```
delta = 1e-6
key, k_i, k_o = random.split(key, 3)

optimize_kl_args = dict(
    likelihood=lh,
    position_or_samples=jft.Vector(lh.init(k_i)),
    n_total_iterations=3,
    n_samples=5,
    key=k_o,
```

(continues on next page)

(continued from previous page)

```
    draw_linear_kwargs=dict(
        cg_name=None,
        cg_kwargs=dict(absdelta=delta * jft.size(lh.domain), maxiter=300),
    ),
    nonlinearly_update_kwargs=dict(
        minimize_kwargs=dict(
            name=None,
            xtol=delta,
            cg_kwargs=dict(name=None),
            maxiter=5,
        )
    ),
    kl_kwargs=dict(
        minimize_kwargs=dict(
            name=None, xtol=delta, cg_kwargs=dict(name=None), maxiter=35
        )
    ),
    sample_mode="nonlinear_resample",
    callback=callback,
)
```

By setting the `sample_mode` of the `optimize_kl` to `nonlinear_resample`, we will use the geoVI algorithm in the inference process. Furthermore, we must now set the `nonlinearly_update_kwargs` in the `optimize_kl`. Similar to the minimizers introduced in the notebook `1_inference`, these set the parameters for a Newton conjugate gradient scheme to calculate geoVI samples.

Furthermore, we can make a function call after each VI iteration by specifying the' callback' argument. As an example, like we did here, we can repeatedly compute plots of specific quantities during the inference.

```
samples, state = jft.optimize_kl(**optimize_kl_args)
```

```
OPTIMIZE_KL: Starting 0001
```

```
N: Iteration Limit Reached
```

```
N: Iteration Limit Reached
```

```
N: Iteration Limit Reached
```

```
N: Iteration Limit Reached
```

```
N: Iteration Limit Reached
```

```
N: Iteration Limit Reached
```

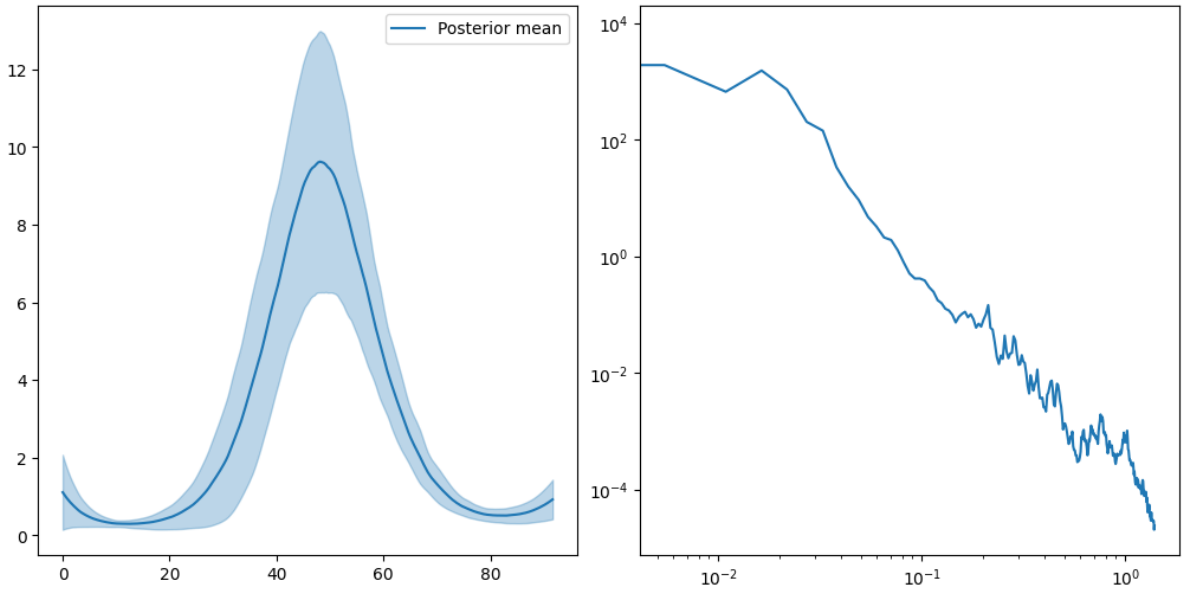
```
N: Iteration Limit Reached
```

```
N: Iteration Limit Reached
```

```
N: Iteration Limit Reached
```

```
N: Iteration Limit Reached
```

```
OPTIMIZE_KL: Iteration 0001 E:+6.2997e+02
OPTIMIZE_KL: #(Nonlinear sampling steps) (5, 5, 5, 5, 5, 5, 5, 5, 5, 5)
OPTIMIZE_KL: #(KL minimization steps) 34
OPTIMIZE_KL: Likelihood residual(s):
'reduced Chi²:    0.98±    0.51, avg:   -0.061±    0.21, #dof:    256'

OPTIMIZE_KL: Prior residual(s):
signal_asperity         :: 'reduced Chi²:    0.94±    0.95, avg:     -0.8±    0.55,
 ↪ #dof:      1'
signal_flexibility      :: 'reduced Chi²:     1.4±     1.5, avg:    +0.77±    0.89,
 ↪ #dof:      1'
signal_fluctuations     :: 'reduced Chi²:     1.2±     1.3, avg:    +0.49±    0.95,
 ↪ #dof:      1'
signal_loglogavgslope   :: 'reduced Chi²:     1.1±     1.0, avg:    +0.92±    0.53,
 ↪ #dof:      1'
signal_spectrum         :: 'reduced Chi²:    0.98±   0.041, avg:    -0.01±   0.044,
 ↪ #dof:    510'
signal_xi               :: 'reduced Chi²:     1.1±   0.081, avg:   -0.017±   0.037,
 ↪ #dof:    512'
signal_zeromode         :: 'reduced Chi²:    0.87±     0.9, avg:    -0.38±    0.85,
 ↪ #dof:      1'
```




```
OPTIMIZE_KL: Starting 0002
```

```
N: Iteration Limit Reached
```

```
N: Iteration Limit Reached
```

```
N: Iteration Limit Reached
```

```
N: Iteration Limit Reached
```

```
N: Iteration Limit Reached
```

```
N: Iteration Limit Reached
```

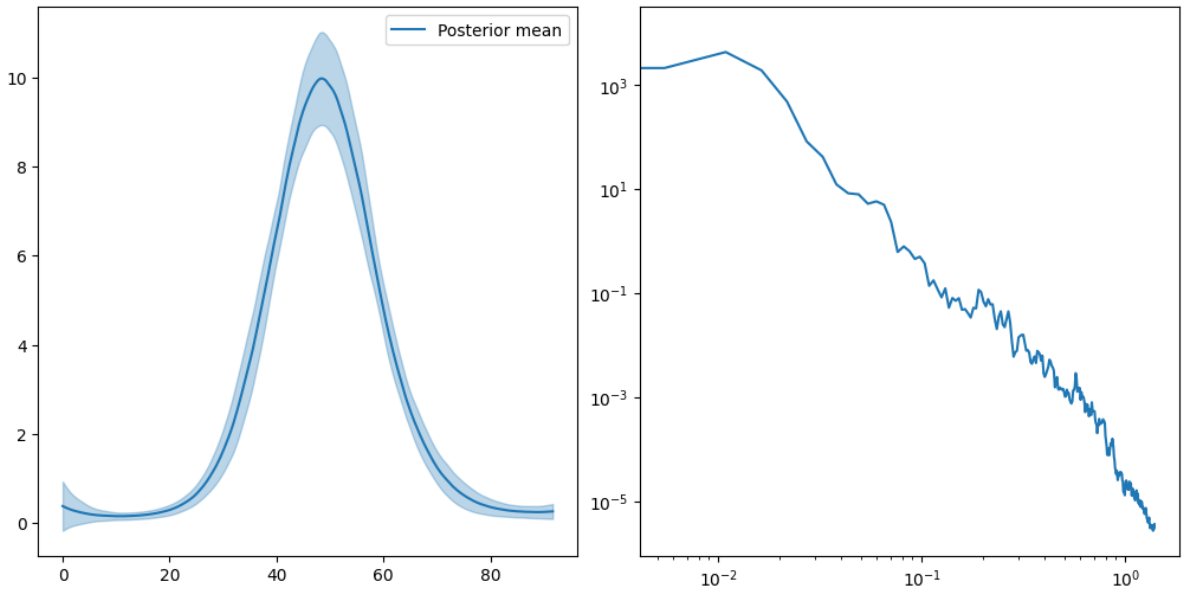
```
N: Iteration Limit Reached
```

```
N: Iteration Limit Reached
```

```
N: Iteration Limit Reached
```

```
N: Iteration Limit Reached
```

```
OPTIMIZE_KL: Iteration 0002 E:+6.1041e+02
OPTIMIZE_KL: #(Nonlinear sampling steps) (5, 5, 5, 5, 5, 5, 5, 5, 5, 5)
OPTIMIZE_KL: #(KL minimization steps) 12
OPTIMIZE_KL: Likelihood residual(s):
'reduced Chi²:    0.74±   0.064, avg:   -0.055±   0.019, #dof:    256'

OPTIMIZE_KL: Prior residual(s):
signal_asperity          :: 'reduced Chi²:    0.49±    0.57, avg:     -0.5±    0.49,
 ↪ #dof:      1'
signal_flexibility       :: 'reduced Chi²:    0.51±    0.56, avg:    +0.56±    0.45,
 ↪ #dof:      1'
signal_fluctuations      :: 'reduced Chi²:     4.3±     4.3, avg:     +1.7±     1.2,
 ↪ #dof:      1'
signal_loglogavgslope    :: 'reduced Chi²:     1.4±     1.8, avg:   +0.086±     1.2,
 ↪ #dof:      1'
signal_spectrum          :: 'reduced Chi²:     1.0±   0.054, avg:   -0.014±   0.043,
 ↪ #dof:    510'
signal_xi                :: 'reduced Chi²:     1.1±   0.075, avg:   -0.016±   0.035,
 ↪ #dof:    512'
signal_zeromode          :: 'reduced Chi²:     1.2±     1.3, avg:    +0.41±     1.0,
 ↪ #dof:      1'
```




```
OPTIMIZE_KL: Starting 0003
```

```
N: Iteration Limit Reached
```

```
N: Iteration Limit Reached
```

```
N: Iteration Limit Reached
```

```
N: Iteration Limit Reached
```

```
N: Iteration Limit Reached
```

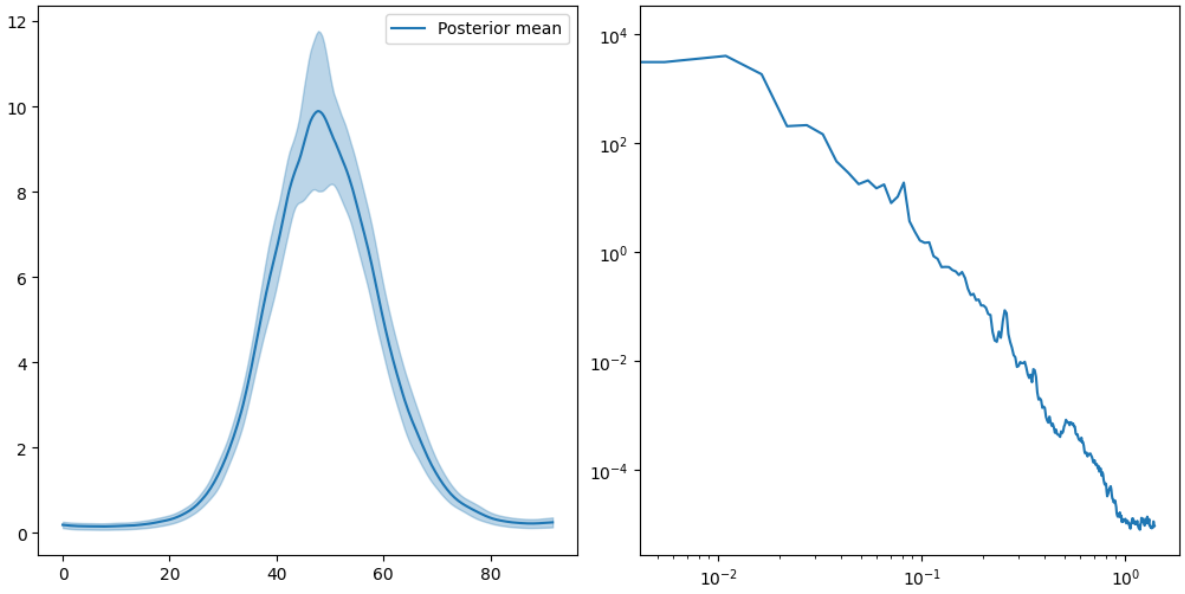
```
N: Iteration Limit Reached
```

```
N: Iteration Limit Reached
```

```
N: Iteration Limit Reached
```

```
N: Iteration Limit Reached
```

```
OPTIMIZE_KL: Iteration 0003 E:+5.8991e+02
OPTIMIZE_KL: #(Nonlinear sampling steps) (5, 5, 5, 5, 5, 5, 5, 5, 5, 5)
OPTIMIZE_KL: #(KL minimization steps) 19
OPTIMIZE_KL: Likelihood residual(s):
'reduced Chi²:    0.69±   0.076, avg:    -0.07±   0.052, #dof:    256'

OPTIMIZE_KL: Prior residual(s):
signal_asperity         :: 'reduced Chi²:     1.4±     1.4, avg:   +0.013±     1.2,
 ↪ #dof:      1'
signal_flexibility      :: 'reduced Chi²:     1.4±     1.5, avg:    +0.42±     1.1,
 ↪ #dof:      1'
signal_fluctuations     :: 'reduced Chi²:     5.0±     4.0, avg:     +2.0±    0.97,
 ↪ #dof:      1'
signal_loglogavgslope   :: 'reduced Chi²:    0.74±    0.88, avg:   -0.053±    0.86,
 ↪ #dof:      1'
signal_spectrum         :: 'reduced Chi²:    0.99±   0.052, avg:   -0.016±   0.032,
 ↪ #dof:    510'
signal_xi               :: 'reduced Chi²:     1.0±    0.03, avg:   -0.015±   0.036,
 ↪ #dof:    512'
signal_zeromode         :: 'reduced Chi²:     1.3±     1.4, avg:    +0.72±    0.87,
 ↪ #dof:      1'
```



6.3 Summary

This notebook introduces how to implement the log-normal Poisson process in `NIFTy`. It presents how to preprocess event count data for a `jft.Poissonian` likelihood. Furthermore, it provides an example of a use case of the correlated field and why we should use the Geometric Variational Inference (geoVI) algorithm. The notebook also showed how to set up `optimize_kl` to use geoVI.

6.4 Appendix

The given data was generated by the rate function

$$\rho(t) = A\,\mathrm{e}^{-\frac{1}{2}\left(\frac{t-t_0}{w}\right)^2}$$

and generate data from it with the Lewis-Shedler (see https://bookdown.org/rdpeng/timeseriesbook/simulation-and-prediction.html for a short summary on it) algorithm to simulate an inhomogeneous process. This gives us a list of events distributed over time.

```
def rho(t, c):
    t0 = c[0]
    w = c[1]
    A = c[2]
    return A * jnp.exp(-((t - t0) ** 2) / (2 * w**2))
```

```
def generate_poisson_data(key, f, f_max, T):
    t = 0
    events = []

    while t <= T:
        key, k0 = random.split(key, 2)

        u = random.uniform(k0)
```

(continues on next page)

(continued from previous page)

```
        del_t = -jnp.log(u) / f_max

        events.append(t)
        t += del_t

    events = jnp.array(events)

    v = random.uniform(key, events.shape)

    p = f(events) / f_max
    mask = v <= p

    return events[mask]


rho_max = 10
T = 100
c = (50, 10, rho_max)


simulated_events = generate_poisson_data(rng_key, partial(rho, c=c), rho_max, T)
```