Hamiltonian Monte Carlo Sampling for Fields

Christoph Lienhard  Torsten A. Enßlin

Abstract

HMCF “Hamiltonian Monte Carlo for Fields”, is a software add-on for the NIFTy “Numerical Information Field Theory” framework implementing Hamilton Monte Carlo (HMC) sampling in Python. HMCF as well as NIFTy are designed to address field inference problems especially in – but not limited to – astrophysics. They are optimized to deal with the typically high number of degrees of freedom as well as their correlation structure.

HMCF adds an HMC sampler to NIFTy that automatically adjusts the many free parameters steering the HMC sampling machinery such as integration step size and the mass matrix according to the requirements of field inference. Furthermore, different convergence measures are available to check whether the burn-in phase has finished. Multiprocessing in the sense of running individual Markov chains (MC) on several cores is possible as well. A primary application of HMCF is to provide samples from the full field posterior and to verify conveniently approximate algorithms implemented in NIFTy.

Keywords: Python, field inference, Hamiltonian sampling.

1. Introduction

1.1. Purpose and Features

HMCF implements a Hamiltonian Monte Carlo (HMC) sampler (Duane, Kennedy, Pendleton, and Roweth 1987) for the NIFTy (“Numerical Information Field Theory”, Steininger, Dixit, Frank, Greiner, Hutschenreuter, Knollmüller, Leike, Porqueres, Pumpe, Reinecke et al. (2017); Reinecke (2018)) framework. It is available for Python3 on Unix-like systems.

Originally used for simulating particle dynamics in high energy physics, HMC sampling is essentially an advanced Metropolis-Hastings algorithm (Hastings 1970) and thereby a Markov Chain Monte Carlo (MCMC) method. It is used to generate samples with arbitrary probability distributions \( P(x) \) as long as a so-called energy \( H(x) := −\log P(x) \) is well defined and the gradient \( \frac{\partial H}{\partial x} \) is available. The main advantages of HMC over classical Metropolis-Hastings
are a much higher acceptance rate of proposed samples and less correlation between samples. HMC gained much attention in recent years due to its applications in machine learning and image reconstruction.

**HMCF** implements such an HMC sampler for the **NIFTy** environment. **NIFTy** is a Python package developed for computational work with information field theory (IFT, Enßlin, Frommert, and Kitaura (2009); Enßlin and Weig (2010)). IFT extends classical probability theory onto functional spaces. **NIFTy** deals with all the technical difficulties arising when working with IFT on computers. A main advantage is the resolution-independent calculation of statistical estimates on large data sets. **NIFTy** is currently mainly used in astrophysical image reconstruction (Selig, Vacca, Oppermann, and Enßlin 2015; Junklewitz, Bell, Selig, and Enßlin 2016; Buscombe 2016; Pumpe, Reinecke, and Enßlin 2018).

With **HMCF**, Bayesian models already implemented in **NIFTy** can easily be reused for an HMC sampling approach. This can help estimating the impact of approximations present in other approaches, or enable tackling entirely new problems.

At the heart of **HMCF** lies the **HMCSampler** class which constructs an HMC sampler based on a predefined **NIFTy** **Energy** class containing the Bayesian model of interest. It incorporates the field’s probability distribution constrained by data and other prior information. **HMCSampler** samples from this probability by setting up a stochastic dynamical system with this energy as a potential energy. **HMCF** uses multi-processing in that individual Markov chains use separate cores. The sampler is capable of deciding on its own when the Markov chains have reached the typical set of possible field configurations (also known as the burn-in phase has finished) by calculating a convergence measure and the so-called mass matrix. The latter is a very important tuning parameter in every HMC sampler. The user can decide how the convergence measure is calculated choosing from predefined strategies but, with a little bit of programming skills, can also define new strategies. The leapfrog integration step size is adjusted during burn in to meet a user defined acceptance rate. Again the adjusting strategy can be chosen from a predefined set of strategies. **HMCF** also provides the HMC mass matrix. It can be readjusted several times during burn in or set by the user in the beginning (or both).

**HMCF** is optimized to ease the work with HMC sampling. All of the above can be done in only a few lines of code if a well-defined **NIFTy** **Energy** class is available.

### 1.2. Comparison to other Packages

There are many software packages for HMC sampling available in many different languages. But unlike **HMCF** most packages are static in that they use in general the identity as the mass matrix or need a mass matrix specified in the beginning. Since especially in high-dimensional cases a good mass matrix estimation is crucial for a successful sampling process we will concentrate on packages which estimate the mass matrix.

A very popular cross-platform package for HMC is **Stan** (Stan Development Team 2017). **Stan** provides interfaces for R, Python, shell, MATLAB, Julia, Stata, Mathematica and Scala. Its biggest advantage is the C++ back-end which makes it by far the fastest sampler if the same parameters such as integration step size and mass matrix are chosen. Another notable advantage over **HMCF** is an implementation of the no-u-turn sampler (NUTS, Hoffman and Gelman (2014)) which can be seen as an extension to the standard HMC approach.

In **Stan** the mass matrix is set to the identity initially, but is recalculated from the generated samples during the burn-in phase. The mass matrix can but does not have to be restricted to
a diagonal matrix. **HMCF** differs in that additionally the user is able to define an own mass matrix which can be an advantage in some cases (see e.g. Taylor, Ashdown, and Hobson (2008)). The Stan developers announced such a feature in future versions, though. Using the samples generated by the initial chain itself involves the risk of having highly correlated samples in case the sampler was malfunctioning due to the usage of the initial mass matrix. To avoid this, **HMCF** uses samples drawn from the curvature of the full posterior at the current position to reevaluate the mass matrix. We found this approach to be much more efficient. Reevaluated mass matrices are always diagonal in **HMCF** but since it is targeted at high-dimensional problems where explicitly defined full matrices are not feasible this is not really a disadvantage. Furthermore, more recent **NIFTy** based algorithms use as field variables harmonic space degrees of freedom (Knollmüller, Steininger, and Enßlin 2017) which fits better to a mass matrix being diagonal in these field parameters.

Another important package for HMC sampling in Python is **pyMC3** (Salvatier, Wiecki, and Fonnesbeck 2018). **pyMC3** provides a huge variety of different samplers among other functions for statistical applications. When it comes to the HMC sampler in **pyMC3** the main difference to **HMCF** is that the mass matrix is again evaluated based on the samples of the Markov chain itself which might be problematic as described in the paragraph above. Again **pyMC3** has a NUTS feature.

Apart from that the main advantage of **HMCF** is that it is easy to use for already written algorithms in **NIFTy** and its optimization for high-dimensional statistical problems.

### 1.3. Structure of this Paper

This introduction is followed by a short installation guide. In section 3 we give an introduction to HMC sampling on a theoretical / mathematical level after which we introduce the **HMCF** module’s classes, methods and functions in great detail. In section 5 we describe the module’s demos with a theoretical background and code examples as well as exemplary results. This document ends with a short summary in section 6 on why there is a need for a distinct **HMCF** package.

### 2. Installation

#### 2.1. Dependencies

**HMCF** relies on the following other Python packages:

**NumPy** : The very basic Python package for numerical analysis on multidimensional arrays.

**SciPy** (Oliphant 2007) : Library implementing advanced algorithms and functions in Python.

**h5py** : A Python wrapper for the HDF5 file format.

**NIFTy** (4.1.0 or newer, Steininger et al. (2017); Reinecke (2018)) : A package for statistical field inference problems.

**matplotlib** (Hunter 2007), optional : A package for producing figures. Necessary for the **HMCF tools** sub-package.
PyQt5 optional: Necessary for the tools sub-package.

NIFTy supports multi-processing in many calculations via mpi4py (Dalcin, Paz, and Storti (2005)) but HMCF needs to restrict each individual Markov chain to one core. If mpi4py is already installed the user should switch multi-processing off by setting the environment variables MKL_NUM_THREADS and OMP_NUM_THREADS to 1 in a terminal:

\[
\begin{align*}
\text{export MKL_NUM_THREADS} &= 1 \\
\text{export OMP_NUM_THREADS} &= 1
\end{align*}
\]

2.2. Installing via Git

Installing HMCF is possible via

\[
\text{pip install git+https://gitlab.mpcdf.mpg.de/ift/HMCF}
\]

3. Problem Description

The goal of most statistical problems is to find expectation values given a distribution $P$.

\[
\langle f(x) \rangle_{P(x)} = \int f(x) P(x) dx
\]

But especially in high dimensional cases the integral may become intractable. One approach to circumvent this problem is to use Markov chain Monte Carlo (MCMC) methods. All MCMC methods have in common that samples $\{x_i\}$ from the distribution $P(x)$ are generated to approximate expectation values as

\[
\langle f(x) \rangle_{P(x)} \approx \frac{1}{N} \sum_{i=1}^{N} f(x_i)
\]

3.1. Metropolis-Hastings Algorithm

The Metropolis-Hastings algorithm (Hastings (1970)) is a MCMC algorithm and a generalization of the Metropolis algorithm (Metropolis, Rosenbluth, Rosenbluth, Teller, and Teller (1953)) to arbitrary distributions.

For a distribution $P$ it generates a Markov chain in two steps:

1. sample a new state according to a proposal distribution $g$

2. accept or reject the proposal with a probability $P_A$

where the acceptance probability $P_A$ for the new step $x'$ given the last position $x$, is defined as

\[
P_A(x'|x) = \min \left( 1, \frac{P(x') g(x'|x)}{P(x) g(x|x')} \right).
\]
This approach defines a valid transition probability for a Markov process with stationary distribution $P$. It has the advantage that it is not necessary to calculate normalization constants of $P$.

However, for high dimensional problems this approach is often problematic since it is far more probable to propose samples that are less probable and therefore are rejected frequently.

### 3.2. Hamilton Monte Carlo

The Hamilton Monte Carlo (HMC) approach (first introduced by Duane et al. (1987), good summaries: Neal et al. (2011); Betancourt (2017)) uses a variation of the Metropolis-Hastings Algorithm trying to tackle the problem of high-dimension. The main idea is to describe the Markov process as a physical Hamiltonian time evolution, and thereby use long-known facts from classical mechanics concerning the dynamics of this system.

First of all the desired probability distribution $P$ is regarded as a potential $\Psi$

$$\Psi(x) := -\log(P(x)) \quad (4)$$

together with a new random ‘momentum’ variable $p$ with covariance $M$ this mimics a full Hamiltonian:

$$H(x, p) = \frac{1}{2} p^\top M^{-1} p + \Psi(x) \quad (5)$$

Note that marginalizing $p$ from the joint distribution

$$P(x, p) \propto \exp(-H(x, p)) \quad (6)$$

leaves us with the original distribution $P(x)$.

To draw a sample we start in our current state $x$ and draw a momentum $p$ from its normal distribution with covariance $M$. With the Hamiltonian given, the system evolves in time according to Hamilton’s equations of motion

$$\dot{x}_i = \frac{\partial H}{\partial p_i} = M^{-1} p \quad (7)$$

$$\dot{p}_i = -\frac{\partial H}{\partial x_i} = \frac{\partial \Psi}{\partial x_i} \quad (8)$$

for $i = 1, \ldots, n$ and $n$ being the dimension of the problem.

In practice often a simple leapfrog algorithm is used to solve these dynamical equations in time. This algorithm has the advantage of being reversible and therefore leaves the target distribution invariant.

$$p \left( t + \frac{\epsilon}{2} \right) = p(t) - \epsilon \left. \frac{\partial \Psi}{\partial x} \right|_{x(t)} \quad (9)$$

$$x(t + \epsilon) = x(t) + \epsilon M^{-1} p$$

$$p(t + \epsilon) = p \left( t + \frac{\epsilon}{2} \right) - \frac{\epsilon}{2} \left. \frac{\partial \Psi}{\partial x} \right|_{x(t+\epsilon)}$$

After time $T = \epsilon N$, where $N$ is the number of integration steps, the system is in a new state $(x', p')$. 
Drawing the momentum and evolving the system in time can be thought of as an elaborate Metropolis-Hastings proposal step. The new state is accepted with probability

$$P_A = \min(1, \exp(-\Delta E))$$

(10)

where $$\Delta E = H(x', p') - H(x, p)$$, which is the same acceptance probability as introduced in equation (3).

In theory $$\Delta E = 0$$ since Hamilton’s equations preserve the energy. But in practice numerical errors make this acceptance step necessary. Because of the energy-preserving property of the Hamiltonian dynamics the acceptance rate is much higher for HMC sampling than for ordinary random walk Metropolis-Hastings algorithms especially in high dimensional cases. Despite its advantages HMC introduces a new set of challenges, especially finding a good covariance $$M$$ for $$p$$. There are several approaches but one very popular strategy is to use the inverse of the covariance of the potential $$\Psi$$ (Betancourt 2017):

$$M^{-1} = \mathbb{E} \left( (x - \mu)(x - \mu)^\top \right)$$

(11)

with $$\mu$$ being the mean value. However, in specific cases other approaches might be better, e.g. as documented in Taylor et al. (2008).

4. Software Structure

Since HMCF can be considered to be an add-on to NIFTy we will give a short introduction to NIFTy followed by a detailed description of HMCF itself.

4.1. The NIFTy Package

NIFTy is a Python package for easing implementation of field inference algorithms. It introduces a variety of classes among which the most important ones are

- **Domain**, containing size and geometry (if there is one present) of the underlying problem.
- so-called **Fields** as the main data carriers, containing the field values in an numpy ndarray and also the **Domain** over which these values live.
- **LinearOperators** are explicitly or implicitly defined linear operators acting on **Fields**.
- and most important for HMCF the **Energy** class

The energy $$H$$ of a field $$s$$ is a functional assigning a value to the field. In IFT the field energy is defined as the negative logarithm of the joint probability of data $$d$$ and field $$s$$.

$$H(d, s) = -\log P(d, s)$$

(12)

This is of great use in many inference algorithms. The NIFTy **Energy** class provides the structure for an efficient implementation of such an energy.

For consistency reasons the NIFTy **Energy** is defined only at one position in the functional space of $$s$$ but has a class method **at(new_position)** returning an **Energy** instance at
new_position (which is a Field instance). An Energy instance provides the value of the energy at the current position as well as the gradient (a Field instance) and its curvature (a LinearOperator instance).

Given an arbitrary distribution $P(d, s)$, the user can define the corresponding energy as child class of Energy as long as he provides the mentioned properties and the at method.

4.2. The HMCSampler Class

The HMCF package is optimized for a fast HMC implementation for a NIFTy Energy class. At its heart lies the HMCSampler class handling the whole sampling process. It is able to run several Markov chains on different CPUs using the Python multiprocessing module. The samples are saved to an HDF5-file which is generated every time a new run is initialized and can be loaded back as needed via the package’s own load function. During the burn-in phase HMCSampler takes care of adjusting the integration step size $\epsilon$ (see equation (9)) such that a user defined acceptance rate is reached, as well as setting and possibly reevaluating the covariance mass matrix $M$. After a run has finished the mean of the samples is calculated.

Of course in practice one may want to fine-tune some of the specific features and parameters implemented in HMCF. This section is dedicated to introduce and explain those.

Instantiating

An instance of HMCSampler is created with the following arguments of which only the first is mandatory:

potential : NIFTy Energy
The HMC potential $V(s)$. Also defines the domain on which the sampling takes place through its position attribute. The potential $V(s)$ is usually identified with $H(d, s)$.

sample_transform : func, optional
In some cases it is preferable to sample a field not in its position space, but in another domain, such as in its harmonic space representation, or maybe even in a domain where there is no linear transformation to the position space. To ensure correct calculation of expectation values the samples are transformed by sample_transform before being saved to disk.

num_processes : int
Number of cores involved in the sampling process. This is equal to the number of individual Markov chains started when the instance method run is called.
Default: 1

sampler_dir_path : str
A path where the HDF5-file containing the samples is going to go.
Default: a new folder called “samples” in the ‘__main__’ script’s directory

Running the Sampling Process

In principle the sampling process can be started immediately after creating an instance of HMCSampler by calling the run() method with the following arguments of which again only the first is mandatory.
HMC Sampling for Fields

num_samples : int
   Number of samples to be drawn per chain after burn in.

max_burn_in : int, optional
   Maximum number of steps for the chain to converge before it is forced into sampling mode. If no value is stated, forced transition is not going to happen. In this case the chain will only start the actual sampling process if it has converged.

convergence_tolerance : float, optional
   If the convergence measure for the sampling process falls below this value, the chain is assumed to have converged and starts with the actual sampling process. If no value is stated the tolerance property of the HMCSampler property convergence is used. The default value for said property is 1. For more on this see section 4.4.

target_acceptance_rate : float, optional
   Value between 0. and 1., stating what ratio of accepted / rejected samples is preferred. The integration step size is adjusted during burn-in to approximately match this ratio. If not stated the corresponding property of the epsilon property is used (for which the default value is 0.8).

mass : NIFTy EndomorphicOperator, optional
   HMC mass matrix used during sampling (or until it is reevaluated). For more on the mass matrix see section 4.6. If no mass is given, an identity matrix is used (at least as initial guess).

x_initial : NIFTy Field or list of NIFTy Fields, optional
   Starting point(s) for the HMC sampler. If more than one Markov chain needs to be initialized they get their respective initial positions by iterating through the list. The list does not have to have the same length as the number of chains. If there are more chains than elements in the list, some starting positions are reused for the additional chains. If only a Field is given, all chains get the same initial position. If no initial field is passed, a random sample is drawn from a Gaussian distribution centered at the position of the Energy instance given to the constructor of HMCSampler with the Energy’s curvature as covariance.

Getting the Results of an HMC-Run

After a run has finished, the sampled mean as a NIFTy Field is accessible via the instance’s property ‘mean’.

```python
In [1]: hmc_sampler = HMCSampler(nifty_energy, num_processes=5)
In [2]: hmc_sampler.run(200)
In [3]: hmc_sampler.mean
Out[3]: nifty4.Field instance
   - domain = DomainTuple, len: 1
      RGSpace(shape=(256, 256), distances=(0.5, 0.5),
      harmonic=True)
   - val = array([[ 0.63, -0.16, ..., 1.04, -0.64],
                   ...,
                   [ 0.03 , 0.02, ..., 1.22, 0.21 ]])
```
Accessing the values of the samples is possible via calling the `samples` property. It consists of a $2 + n$ dimensional `numpy` `ndarray` where the first dimension represents the different Markov Chains, the second dimension represents the individual samples and the other $n$ dimensions represent the value of the sampled `NIFTy` Fields. Remember though that calling this will load all samples into memory which might crash the process if not enough memory is available.

```python
In [4]: all_samples = hmc_sampler.samples
Out[4]: loaded chain 00
Out[4]: loaded chain 01
Out[4]: loaded chain 02
Out[4]: loaded chain 03
Out[4]: loaded chain 04
In [5]: all_samples.shape
Out[5]: (5, 200, 256, 256)
```

### Attributes and Properties of `HMCSampler`

The `HMCSampler` has a number of other properties and attributes which are mostly used for fine-tuning the sampling process. These are:

- **potential**: `NIFTy` Energy, read-only
  
  The potential $\Psi$ for the HMC sampler.

- **sampler_dir**: str
  
  Setting or getting the path to the directory where the sample-files are stored. Corresponds to the parameter of the same name passed in the constructor of `HMCSampler` class.

- **save_burn_in_samples**: bool
  
  Whether or not to save the samples generated during burn-in phase to disk. Be aware of the fact that if set to `True` (default value) together with a high or non-existent `max_burn_in` parameter (in the constructor of `HMCSampler` class) could fill your hard drive.
  
  Default: True

- **convergence**: `HMCF` Convergence
  
  For choosing how to calculate the convergence measure (see section 4.4).
  
  Default: `HansonConvergence` if `num_processes == 1` else `GelmanRubinConvergence`

- **epsilon**: `HMCF` Epsilon
  
  For choosing how to adjust the integration step size parameter during burn-in (see section 4.5).
  
  Default: `EpsilonSimple`

- **mass**: `HMCF` Mass
  
  For choosing how to handle the HMC mass during sampling. For more on this see section 4.6.

- **display**: `HMCF` Display
  
  For choosing how to display the progress of the sampling process (see section 4.7)
  
  Default: `LineByLineDisplay`
n_limits: list of ints
   To avoid periodic trajectories the number of leapfrog integration steps is randomized.
   n_limits defines the range from which the number of integration steps is drawn uniformly.
   Default: [60, 70]

4.3. The HMCSamplerDebug class

The HMCSamplerDebug class is essentially the same class as HMCSampler with a few additional features related to logging more information about and during the sampling process.

This means in particular, that instantiating the class and running Markov chains works exactly like in the HMCSampler class. The difference lies in additional attributes and properties and a more complex HDF5 file generated during sampling.

The additional logging attributes can be chosen prior to calling the run method by setting the following attributes of HMCSamplerDebug:

log_raw_samples: bool
   In the HMCSampler class, samples are transformed by a function given as optional parameter sample_transform in the constructor. If log_raw_samples is True the untransformed samples are saved, too.
   Default: True

log_convergence_measure: bool
   If True the convergence measure is logged for every chain at every step.
   Default: True

log_integration_error: bool
   Whether or not to log the integration error after the burn-in phase is over. This attribute is not saved to the HDF5 file but rather used to calculate a mean integration error by integration step. This can be useful in finding a relation between number of steps and integration error (after reaching a typical set). This slows down sampling significantly.
   Default: True

log_potential_values: bool
   Whether or not to log the value of the potential at each step for each chain.
   Default: True

log_gradients: bool
   Whether or not to log the values of the gradient at each step for each chain.
   Default: True

After a run the additionally logged attributes can either be loaded by calling the HMCF load function with the appropriate attr_type parameter (see 4.8) or by calling one of the following HMCSamplerDebug properties:

raw_samples: numpy ndarray
   Same as the samples property of the HMCSampler class but with the ‘raw’, untransformed samples.
convergence_measure : numpy ndarray
   Same structure as the samples property of the HMCSampler class but instead of the
   values for each point, their convergence measures are returned.

integration_error : numpy ndarray
   A one-dimensional array with size n_limits[1]+1. Contains the mean integration error
   after each integration step. This can be useful in finding a relation between number of
   steps and integration error (after reaching a typical set).

potential_values : numpy ndarray
   A two-dimensional array containing the value of the potential for every chain at every
   step. The first dimension represents the chain and the second dimension the corre-}
{sponding step.

gradients : numpy ndarray
   Same structure as the samples property of the HMCSampler class but instead of the
   values for each point, the values of the gradients at that position are returned.

4.4. Convergence

The Convergence class handles everything related to the convergence of the Markov chain(s).
In principle a chain in HMCF has converged if a ‘convergence measure’ calculated for each
degree of freedom in the sampled NIFTy Field drops below a given ‘tolerance’.
Additionally HMCF implements intermediate steps of convergence via so-called convergence
‘levels’. For the time being their main purpose is to define a time where the HMC mass is
reevaluated during burn-in (See also section 4.6). A chain is said to have converged with
respect to its current convergence level, if

$$\max(\text{measure}) < \text{tolerance} \cdot 10^{\text{level}}$$

In other words: The level is the number of digits the decimal separator of the tolerance is
shifted to the left.

The idea behind this is to decrease the level by one each time an intermediate convergence is
reached, while at that point recalculating the mass matrix.
If the level drops to zero, equation (13) simplifies to max(measure) < tolerance and the next
time this requirement is met, the Markov chain has finished the burn-in phase.
It remains to explain how the convergence measure is calculated. There are several different
approaches implemented in HMCF as child classes of the Convergence class. Choosing one
of them is done by setting the convergence property of the HMCSampler class with one of the
Convergence’s child classes, e.g.:

```python
hmc_sampler.convergence = GelmanRubinConvergence
```

For now there are four different possibilities:

MeanConvergence This rather simple way of calculating the convergence needs at least two
Markov chains. It compares the mean of the samples from all chains (total_mean) to the
mean of each individual chain (\texttt{chain\_means}). The measure is defined as \texttt{abs(chain\_mean / total\_mean - 1.)} such that it fulfills the non-negativity and the identity of indiscernibles criteria for metrics. It proves to be rather unstable if e.g. the total mean is close to zero.

\textbf{VarConvergence} Very similar to \textbf{MeanConvergence} only with the variances of individual chains and all chains. Measure is equal to \texttt{abs(chain\_var / total\_var - 1.).}

\textbf{HansonConvergence} So far the only convergence measure which can be used even if there is only one chain. It follows Hanson (2001) (Again: the measure is the absolute value of the ratio minus one).

\textbf{GelmanRubinConvergence} An implementation of the among MCMC folks very popular Gelman and Rubin convergence criteria Gelman and Rubin (1992) (Again: the measure is the absolute value of the ratio minus one).

\textit{Attributes and Properties of Convergence}

Regardless of which \texttt{Convergence} child class has been used additional features can be set via its class properties, e.g. the \texttt{locality} property which defines the number of recent samples considered when calculating the \texttt{convergence} (see below for details):

\begin{verbatim}
    hmc_sampler.convergence = GelmanRubinConvergence
    hmc_sampler.convergence.locality = 200
\end{verbatim}

For the common user the following properties are the most important ones:

\texttt{locality : int}

The number of recent samples to be considered in calculating the convergence measure. On the one hand this is a form of ‘forgetting’ very old parts of the chain’s trajectory which do not represent the current state of convergence. On the other hand this is necessary because of memory issues i.e. if the burn-in phase takes very long the memory would blow up since every sample ever created has to be available to calculate the measure.

Default: 250

\texttt{tolerance : float}

Equivalent to the \texttt{convergence\_tolerance} parameter of the \texttt{HMCSampler’s run} method. In fact, setting this property as described above has only an effect if the (optional) \texttt{convergence\_tolerance} parameter is not passed to the \texttt{run} method.

In practice the latter approach might be slightly more convenient. If the maximum value of a chain’s convergence measure is below this value the chain is said to have converged and transitions from the burn-in phase into the sampling phase. See also: \texttt{converged} (below)

Default: 1.

The following additional properties of \texttt{Convergence} are mostly only important for \texttt{HMCSampler} itself and not of relevance for the common user:
converged : `numpy ndarray` of bool (1 dim)
Contains the information of whether the individual chains have converged with respect to the following law:

```
converged = measure_max < tolerance * 10**level
```

**measure : `numpy ndarray`**

(1 + n dim)
Represents the value of the measure (calculated dependent on which child class of the `Convergence` class has been used) for each element of the degrees of freedom in the sampled `NIFTy Field`. The first dimension represents the individual chains.

**measure_max : `numpy ndarray`**

(1 dim)
The highest value of the `Convergence` class property `measure` per chain.

**level : `numpy ndarray`**

of int (1 dim)
See class property `converged`. The idea is that after a Markov chain has converged with respect to its current level the level is decreased by one. There are `Convergence` class methods `dec_level` and `inc_level` for decreasing and increasing the level by 1, respectively. For more details on these methods see below.
Setting this property is also possible with a simple `int` which sets the whole `numpy ndarray` to that value.

**quota : `numpy ndarray`**

of float (1 dim)
The ratio of elements in the sampler’s position `NIFTy Field` which have converged with respect to `tolerance` and `level` (i.e. the ‘intermediate’ convergence)

### Additional Methods of Convergence

Internally the convergence levels are decreased and increased by calling

**dec_level(chain_identifier=None)**
decreases the convergence level of `chain_identifier (int)` by one. If `chain_identifier` is `None` the level of all chains is decreased by one. Either way if the level of a chain is already zero it is left unchanged.

**inc_level(chain_identifier=None)**
increases the convergence level of `chain_identifier (int)` by one. If `chain_identifier` is `None` the level of all chains is increased by one.

The convergence level is set under the hood dependent on specific properties of the `Mass` class in the beginning of the `HMCSampler’s run` method.

### 4.5. Epsilon

The $\epsilon$ parameter defines the leapfrog integration step size (equation (9)). In principle the bigger it is the bigger is the integration error $\Delta E$ and thereby the smaller the acceptance rate. To achieve an approximate acceptance rate defined via the `target_acceptance_rate` parameter of `HMCSampler’s run` method, $\epsilon$ has to be adjusted during burn in.
Achieving this requires to find a relation between acceptance rate $r_A$ and the during sampling acquired integration error $\Delta E$. For HMCF we developed the following approximation. Let’s start from an observation about the acceptance probability $P_A$ (see equation (10)). The expected acceptance rate is

$$r_A(\epsilon) = \langle P_A(\Delta E) \rangle_{P(\Delta E|\epsilon)} = \langle \min\left(1, e^{-\Delta E}\right) \rangle_{P(\Delta E|\epsilon)}$$  \hspace{1cm} (14)

where $P(\Delta E|\epsilon)$ is the probability distribution for $\Delta E$ conditioned on $\epsilon$.

To tackle the min function properly we assume that $P(\Delta E) = P(|\Delta E|)P(\text{sgn}(\Delta E))$ i.e. the probability distribution for the sign of $\Delta E$ is not dependent on the absolute value of $\Delta E$. This reflects the plausible situation that errors are symmetrically probable displacements of trajectories in regions of the phase space which are dominated by a potential gradient and not by a minimum. In this case we can further assume that

$$P(\text{sgn}(\Delta E) = 1) = P(\text{sgn}(\Delta E) = -1) = 0$$  \hspace{1cm} (16)

With that, equation (14) can be written as

$$r_A(\epsilon) = \frac{1}{2} \left(1 + \langle e^{-|\Delta E|} \rangle_{P(|\Delta E|\epsilon)}\right) \approx \frac{1}{2} \left(2 - \langle |\Delta E| \rangle \right) + O(\langle |\Delta E| \rangle^2)$$  \hspace{1cm} (17)

where we expanded the exp-function to first order. In practice $r_A$ is given as the target_acceptance_rate and the goal is to achieve an expected value for $\Delta E$ with:

$$\langle |\Delta E| \rangle \overset{!}{=} 2(1 - r_A(\epsilon)) =: \Delta E_{\text{wanted}}$$  \hspace{1cm} (18)

This is the relation that lies at the core of every epsilon-adjusting strategy available in HMCF.

Note that even if $\Delta E$ is negative and therefore acceptance a fact it can still be used to adjust $\epsilon$ to properly match an target_acceptance_rate since only the absolute value is necessary. This is of great use in cases where nearly every step during burn-in produces a negative $\Delta E$ (This happens sometimes if the Markov chains start far off the mean value).

In HMCF the Epsilon class, much like the Convergence class, is just a base class and much more interesting for the common user are its child classes defining exactly how $\epsilon$ is adjusted.

The class also keeps track of how much $\epsilon$ has changed in recent steps and how close the mean value of recent integration errors $\langle \Delta E \rangle$ is to $\Delta E_{\text{wanted}}$. If $\epsilon$ has not changed very much and $\langle \Delta E \rangle \approx \Delta E_{\text{wanted}}$, Epsilon is said to have converged.

If Epsilon has converged its value is locked.

**Available Adjusting Strategies**

EpsilonConst $\epsilon$ stays constant throughout the whole sampling process. The value can be set by setting its val attribute:

```python
hmc_sampler.epsilon = EpsilonConst
hmc_sampler.epsilon.val = 0.005
```
EpsilonSimple $\epsilon$ gets reduced or increased if $\Delta E$ is bigger or smaller than $\Delta E_{\text{wanted}}$ respectively independent of the absolute value of $\Delta E$.

In practice, EpsilonSimple has an attribute change_range (float between 0 and 1, Default: 0.1), which can be set via:

```python
hmc_sampler.epsilon = EpsilonSimple
hmc_sampler.epsilon.change_range = 0.2
```

This attribute is only available in EpsilonSimple. Given the change_range the current value of $\epsilon$ is multiplied by a factor drawn from a uniform distribution $U([a,b])$ where

\[ [a,b] = \begin{cases} 
[1 - \text{change\_range}, 1] & \text{if } \Delta E > \Delta E_{\text{wanted}} \\
[1, 1 + \text{change\_range}] & \text{if } \Delta E < \Delta E_{\text{wanted}}
\end{cases} \]

The randomness is necessary to prevent recurrent behavior if the integration error $\Delta E$ is very sensitive to $\epsilon$.

EpsilonPowerLaw $\epsilon$ gets adjusted just like EpsilonSimple but the change_range is now defined by the relative difference between $\Delta E$ and $\Delta E_{\text{wanted}}$ (EpsilonPowerLaw has no attribute change_range!). Given this class’s attribute power (positive int, Default: 5), set via

```python
hmc_sampler.epsilon = EpsilonPowerLaw
hmc_sampler.epsilon.power = 4
```

the change_range in EpsilonSimple is defined as:

\[
\text{change\_range} = \frac{|\Delta E - \Delta E_{\text{wanted}}|}{\Delta E + \Delta E_{\text{wanted}}}^{\text{power}}
\] (19)

EpsilonPowerLawDivergence In practice working with Poissonian or log-normal distributions on high dimensional spaces the integration error $\Delta E$ proved to be very sensitive to small changes in $\epsilon$. With this class $\epsilon$ is adjusted just like EpsilonPowerLaw with the difference, that in case of a divergent $\Delta E$ (e.g. during integration an overflow occurs) the change_range becomes more sensitive.

A divergence_counter keeps track of the number of times a divergent behavior was detected and the change_range defined in equation (19) gets a prefactor $2^{-\text{divergence\_counter}}$.

EpsilonExponential In this case a simple connection between $\Delta E$ and $\epsilon$ is assumed:

\[
|\Delta E|(\epsilon) = a \cdot \epsilon^b
\] (20)

where $a > 0$ and $b > 0$ are fitting parameters. This assumption is motivated by the fact that $\Delta E$ tends to zero if $\epsilon$ does and diverges for large $\epsilon$. Former ‘measurements’, i.e. sampling steps of $\Delta E$ given $\epsilon$ are used to calculate $a$ and $b$. This approach asks for a rather large value for the locality property (see below). $\epsilon$ is adjusted by rearranging equation (20) such that:

\[
\epsilon_{\text{new}} = \left(\frac{1}{a} |\Delta E_{\text{wanted}}|\right)^\frac{1}{b}
\] (21)
In this case another connection between $\Delta E$ and $\epsilon$ is assumed:

$$|\Delta E|(|\epsilon|) = \frac{a}{(\epsilon_0 - \epsilon)^b} + \text{const} \quad (22)$$

where $a > 0$ and $b > 1$ are again fitting parameters and const is such that $|\Delta E|(|\epsilon = 0|) = 0$. The idea behind this relation is an updated version of EpsilonExponential where there is a finite $\epsilon_0$ for which $\Delta E$ diverges already. $a$ and $b$ are again fitted with former $\Delta E$s given $\epsilon$, whereas $\epsilon_0$ is set to the current value of $\epsilon$ every time a divergent behavior is detected. $\epsilon$ gets adjusted by rearranging equation (22), such that

$$\epsilon_{\text{new}} = \epsilon_0 \left(1 - \left(1 + \frac{\epsilon_0^b \Delta E_{\text{wanted}}}{a}\right)^{-\frac{1}{b}}\right) \quad (23)$$

**Attributes and Properties of Epsilon**

Regardless of which Epsilon class has been used, additional features can be set via its class properties, e.g. the locality property which defines the scope of fitting points for classes like EpsilonExponential and for calculating the convergence measure (see below for details):

```python
hmc_sampler.epsilon = EpsilonPowerLawDivergence
hmc_sampler.epsilon.locality = 20
```

For the common user the following properties and attributes are the most important ones:

- **val** : float
  The (current) value of $\epsilon$. It is possible to use this property to set a good initial guess (although most of the time unnecessary).
  Default: 0.005

- **locality** : int
  The number of recent samples to be considered in calculating the convergence measure of epsilon.
  Default: 10

- **target_acceptance_rate** : float
  Equivalent to the target_acceptance_rate parameter of the HMCSampler’s run method. In fact, setting this property only has an effect if the (optional) target_acceptance_rate parameter is not passed to the run method.
  Default: 0.8

- **convergence_tolerance** : float
  Essentially the same thing as the tolerance property of Convergence (section 4.4) but for epsilon A value > 1 is unreasonable because of how the convergence measure for epsilon is calculated (see below)
  Default: 0.5

- **divergence_threshold** : float
  The value of $\Delta E$ for which the integration is said to have diverged.
  Default: 1E50
epsilon_limits : list of float
Minimum and maximum value for Epsilon. If the adjusting algorithm proposes a value ‘out of range’ the value gets coerced.
Default: [1.E-40, 10]

Under the hood HMCSampler uses the following additional properties of Epsilon:

converged : bool, read-only
Whether or not epsilon has converged or not, i.e. whether the convergence_measure
is smaller than the convergence_tolerance

measure : list of float, read-only
The convergence measure for epsilon contains two values. The first is the relative variance of the value $\epsilon$, the second represents how close the mean value of $\Delta E$ is to $\Delta E_{\text{wanted}}$. If both are smaller than convergence_tolerance, epsilon has converged.

4.6. Mass
Finding an appropriate mass matrix is the most challenging task for a good HMC sampler. HMCF provides the user with the standard evaluation procedure introduced in equation (11) as well as the possibility to define a problem specific mass matrix.

(Re-)evaluation is done by drawing samples \( \{x^{(i)}\} \) from the curvature at the current position of the NIFTy Energy given as potential parameter in the constructor. To keep the complexity of the problem bearable only the diagonal of the mass matrix in equation (11) is calculated and used:

\[
M_{jk} = \delta_{jk} \frac{1}{N - 1} \left( \sum_{i=1}^{N} x_j^{(i)^2} \right)^{-1}
\]

(24)

This also removes the problem that for a non-degenerate mass matrix in \( n \) dimensions at least \( n \) independent samples are required. For typical applications in NIFTy \( n = 10^6..10^8 \) easily.

The idea behind the HMCF Mass class is to easily define a strategy of handling the mass matrix of an HMC process. A main difference to the Epsilon and Convergence classes is that the mass property of HMCSampler class is not supposed to be set. The mass handling strategy is defined via properties of the Mass class. It is possible to reevaluate the mass matrix several times during burn-in phase. In principle the mass matrix is only evaluated if Epsilon has converged. Additionally the chain has to have converged with respect to its current convergence level introduced in section 4.4. As default the identity is used as mass matrix but an initial mass matrix can be evaluated without reaching any level of convergence (Epsilon still has to have converged). By default there is one initial mass evaluation.

get_initial_mass : bool
Whether or not to evaluate an initial mass. Setting this to True/False will increase/decrease reevaluations by 1 respectively.
Default: True

reevaluations : numpy ndarray of int
The number of reevaluations (including the initial one if set) for each chain. Setting is
also possible (and recommended) with a simple int which sets all chains to that int. Default: numpy.ones(num_chains) (i.e. one reevaluation for every chain)

**operator : NIFTy EndomorphicOperator**

The actual mass operator. If there is a problem specific mass operator it can be set with this property before initiating the run. If get_initial_mass is True, setting operator will set get_initial_mass to False and decrease reevaluations by 1. Default: Identity (as NIFTy DiagonalOperator)

**num_samples : int**

Defines the number of samples drawn from the curvature when reevaluating the mass matrix. Default: 50

**shared : bool**

If True reevaluating a mass matrix is done at the mean of all individual chain positions. All chains have to meet the conditions for evaluation mentioned above. Afterwards each chain gets the same mass matrix. If False each chain reevaluates its mass matrix individually if the chain meets the conditions for evaluation. Default: False

### 4.7. Display

Naturally HMC sampling can take quite a while. Disadvantageous settings of parameters might lead to a malfunctioning sampling process. To be able to discover a pathological run can save hours or even days. For this reason HMCF offers a number of display modes for diagnostic purposes. On the other hand displaying indicators of tens of parallel running chains might be overwhelming. The display property of HMCSampler class offers three different possibilities of displaying several indicators during sampling.

Similar to the Epsilon and Convergence classes there are several Display classes which define the three displaying modes.

**Display :** Displays nothing at all. Serves as base class for the two other display classes.

**LineByLineDisplay :** If the property level is set to INFO or below certain indicators are printed at every sampling step as depicted in figure 1a). Otherwise only warnings and errors are printed.

**TableDisplay :** The most advanced version of displaying indicators. A table is generated and dynamically updated, containing information for each chain as depicted in figure 1b). This class relies heavily on the curses Python package and therefore alters the terminal behavior during sampling.

The columns in TableDisplay display the following parameters:
a) LineByLineDisplay with level = DEBUG during sampling

| chain | acc. rate | dEnergy | Convergence | conv. lev meas. | quota | Epsilon value | conv. meas. |
|-------|-----------|---------|-------------|-----------------|-------|---------------|-------------|
| 6     | 0.95      | 5.38E-01| True        | 1 5.18E-01 100.00% | 2.51E-03 True 4.16E-01 |
| 2     | 1.00      | -3.57E-01| True        | 1 5.20E-01 100.00% | 2.24E-03 True 4.53E-02 |
| 3     | 0.95      | 5.44E-02 | True        | 1 5.19E-01 100.00% | 2.37E-03 True 4.94E-01 |
| 4     | 1.00      | 1.89E-05 | True        | 0 5.11E-01 100.00% | 2.22E-03 False 1.00E+00 |
| 5     | 0.95      | -2.45E-01| True        | 1 5.17E-01 100.00% | 2.36E-03 True 2.70E-01 |
| 6     | 0.95      | -8.25E-02| True        | 1 5.18E-01 100.00% | 2.55E-03 True 8.61E-02 |
| 7     | 0.90      | 7.65E-05 | True        | 0 5.11E-01 100.00% | 5.44E-03 False 1.00E+00 |
| 8     | 0.95      | -7.89E-02| True        | 1 5.17E-01 100.00% | 2.28E-03 True 2.05E-01 |
| 9     | 1.00      | -3.31E-05| True        | 0 5.15E-01 100.00% | 2.72E-03 False 1.00E+00 |
| 10    | 0.95      | 4.10E-01 | True        | 1 5.46E-01 100.00% | 2.46E-03 True 3.50E-01 |
| 11    | 0.90      | -1.07E+00| True        | 1 5.16E-01 100.00% | 2.98E-03 True 3.91E-01 |

b) TableDisplay with level = DEBUG during burn in

Figure 1

ch The chain number or ‘identifier’.
acc. rate The acceptance rate for each chain. During burn in this is only the acceptance rate of the last 10 samples since this highly depends on the value of Epsilon. After burn in acc. rate displays the overall acceptance rate.
dEnergy The most recent value of the integration error $\Delta E$.
Convergence Whether the chain has converged with respect to the current convergence level.
conv. The current convergence level.
lev The maximum value of the convergence level of the chain.
meas. The percentage of points in the sampled NIFTy Field which have converged with respect to the current convergence level.
quota The maximum value of the convergence measure.
Epsilon value The current value of Epsilon.
conv. Whether or not Epsilon has converged with respect to its measure.
meas. The maximum value of the Epsilon convergence measure.
**Attributes and Properties of Display**

Regardless of which Display class has been used an additional property can be set: \texttt{level}.

\texttt{level : int}

This is equivalent to the levels introduced by the Python package \texttt{logging}. The value of \texttt{level} has slightly different effects on the different Display classes, but all in all it works like \texttt{logging} as in the lower the level the more information is displayed.

Default: \texttt{logging.INFO}

### 4.8. Additional Functions

\texttt{HMCF} provides a number of additional functions targeted at easing the handling of the HDF5-files generated during sampling. These files have rather cryptic default names of the form ‘runYYYY-MM-DD\_hh-mm-ss.h5’, where Y, M, D, h, m and s represent year, month, day, hour, minute and second digits at the time of the initialization of that run, respectively. There are three functions handling data stored in these files: \texttt{load}, \texttt{load_mean}, \texttt{load_var}

#### The \texttt{load} Function

The \texttt{load} function takes the following arguments of which only the first is mandatory. Keep in mind that for high dimensional problems the amount of data loaded to memory can easily be several GiBytes.

\texttt{path : str}

Either the path to the HDF5 file or a path to the directory where the HDF5 file(s) are stored. If (path) is a directory, the latest ‘run’ file (with respect to the file name) is loaded.

\texttt{attr_type : str}

The ‘attribute’ to load. For files generated with the \texttt{HMCSampler} class possible values are: ‘burn\_in’ and ‘samples’. For ‘burn\_in’ the samples generated during burn in are loaded (of course, this is only possible if the \texttt{HMCSampler} class attribute \texttt{save\_burn\_in\_samples} was set to True). For ‘samples’ the samples generated after burn in are loaded. If the sampling process was done with \texttt{HMCSamplerDebug} additional attributes are available:

- ‘conv\_meas’  The convergence measures at each step for each chain
- ‘pot\_vals’  The value of the potential at each step for each chain
- ‘gradients’  The gradient of the potential at each step for each chain

Default: ‘samples’

\texttt{start : int}

Only loads the samples from step \texttt{start} onward.

Default: 0

\texttt{stop : int, optional}

Only loads the samples up to step \texttt{stop}. Loads until the end if no value is passed
step : int
    Only loads every \( n \)th sample, where \( n \) is given by \texttt{step}.
    Default: 1

In all cases the function returns a \texttt{numpy ndarray}. The exact shape depends on the \texttt{attr_type}
parameter. In the case of \texttt{pot_vals} the \texttt{numpy ndarray} has two dimensions: The first
represents the different chains, the second the different steps. In all other cases the array
has \( 2 + n \) dimensions, where again, the first and second dimension represent chains and
steps respectively. If not all chains have the same number of samples (e.g. \texttt{attr_type} =
\texttt{‘burn_in’}, every chain needs a different number of steps to reach convergence) shorter chains
are filled with \texttt{numpy.nans} in the output array to match the size of the longest chain.

The \texttt{load_mean} Function

The \texttt{load_mean} function calculates the mean value based on the samples saved to a HDF5
file.

\texttt{path} : str
    Either the path to the HDF5 file or a path to the directory where the HDF5 file(s) are
    stored. If \texttt{(path)} is a directory, the latest ‘run’ file (with respect to the file name) is
    loaded.

\texttt{domain} : \texttt{NIFTy Domain}, optional
    If \texttt{domain} is given the function output is a \texttt{NIFTy Field} with domain \texttt{domain}
    and the calculated mean as value.

The function returns the mean value either as \texttt{numpy ndarray} if \texttt{domain} is not given or as a
\texttt{NIFTy Field} if \texttt{domain} is given.

The \texttt{load_var} Function

The \texttt{load_var} function calculates the variance value based on the samples saved to an HDF5
file.

\texttt{path} : str
    Either the path to the HDF5 file or a path to the directory where the HDF5 file(s) are
    stored. If \texttt{(path)} is a directory, the latest ‘run’ file (with respect to the file name) is
    loaded.

\texttt{domain} : \texttt{NIFTy Domain}, optional
    If \texttt{domain} is given the function output is a \texttt{NIFTy Field} with domain \texttt{domain}
    and the calculated variance as value.

The function returns the variance either as a \texttt{numpy ndarray} if \texttt{domain} is not given or as a
\texttt{NIFTy Field} if \texttt{domain} is given.

4.9. Tools

\texttt{Tools} is a sub-module of \texttt{HMCF} which provides two handy functions to evaluate an already
finished sampling process. The one, \texttt{show_trajectories}, provides a GUI to quickly visualize
the individual Markov chains, the other, `get_autocorrelation`, calculates the autocorrelation of the chains.

**The show_trajectories Function**

A simple GUI for visualizing Markov chains based on one- or two-dimensional problems. The GUI is divided into two graphs as displayed in figure 2. The left one represents the underlying problem space i.e. its geometry with either the ‘mean’ value or a summary of the convergence measures during burn in called ‘bad pixel distribution’. The bad pixel distribution is a map displaying essentially a mean value of all the convergence measures at each step in each chain for each degree of freedom (or pixel). The right graph displays the trajectory of the different chains for a selected degree of freedom of the problem. A pixel can be selected by either entering its coordinates in the top row and clicking on show, or by just clicking on it in the left reference picture.

The function takes the following parameters of which only the first is mandatory:

**path : str**

Either the path to an HDF5 file or a path to the directory where the HDF5 file(s) are stored. If (path) is a directory, the latest ‘run’ file (with respect to the file name) in said directory is loaded.

**reference_field : str**

Either 'mean' or 'bad_pixel_distr' defining the left part of the GUI as described

Figure 2: The evaluation GUI for displaying the Markov chain trajectories of selected pixels. On the left the mean value of a 256x256 pixels Wiener filter reconstruction is displayed, on the right the trajectories of the five Markov chains at the pixel coordinates (x=183, y=64) (as stated in the top row).

entering its coordinates in the top row and clicking on show, or by just clicking on it in the left reference picture.
above.
Default: ’mean’

solution : NIFTy Field or numpy ndarray, optional
In case a ‘right’ answer is available (e.g. a mock data example) it can passed here and is displayed in the trajectories graph as horizontal line as additional reference.

start : int
Only loads the samples from step start onward.
Default: 0

stop : int, optional
Only loads the samples up to step stop. Loads until the end if no value is passed

step : int
Only loads every nth sample, where n is given by step.
Default: 1

The get_autocorrelation Function
Calculates the autocorrelation of the samples (after burn in) for a given t, where t is the shift in
\[
\text{auto_corr}[t] = \sum_i x(i)\bar{x}(i + t)
\] (25)

The function takes the following two arguments:

path : str
Either the path to an HDF5 file or a path to the directory where the HDF5 file(s) are stored. If (path) is a directory, the latest ‘run’ file (with respect to the file name) in said directory is loaded.

shift : int
The shift t as described above.
Default: 1

The function returns a 1+n dimensional numpy ndarray where the first dimension represents the different chains and the other n dimensions the dimensionality of the problem.

5. Examples

5.1. Wiener Filter
A rather easy inference problem assuming that a measurement with an instrument provides a data field d with the stochastic equation
\[
d = R s + n
\] (26)
where n is Gaussian white noise (covariance N) and s the signal. R is a linear operator representing the instrument’s response.
If \( s \) has a Gaussian prior with covariance \( S \) one can show that the full posterior is

\[
P(s|d) \propto \exp \left( -\frac{1}{2} s^\top S^{-1} s - \frac{1}{2} (d - Rs)^\top N^{-1} (d - Rs) \right)
\]

\[
\propto \exp \left( -\frac{1}{2} (s - m)^\top D^{-1} (s - m) \right)
\]

where \( D^{-1} = S^{-1} + R^\top N^{-1} R \) and \( m = DR^\top N^{-1} d \) the mean value.

The advantage of choosing this problem is that there exists an analytic solution for the mean value \( m \) and therefore a comparable solution for the HMC sampler.

We assume the covariance \( S \) to be diagonal in the harmonic representation of the space for \( s \) with a power spectrum \( P(k) \):

\[
P(k) \propto \left( 1 + \frac{k}{k_0} \right)^{-4}
\]

where \( k_0 \) defines the correlation length. Furthermore the signal-to-noise ratio (SNR) is 1, meaning that the covariance of \( n \) is roughly as big as the (mean) variance of \( s \) in the position space.

**Implementation in HMCF**

The following code snippets can also be found in the HMCF package’s demo folder in the wiener_filter_demo.py script. A full Wiener filter demo (without HMC sampling) can also be found in the NIFTy package.

The demo works by default on a 64x64 pixel regular space with the covariance of the signal \( s \) being given as described above. This means that there are 4,096 degrees of freedom. For simplicity assume that for the following example \( D, S, R \) and \( N \) are already well defined as NIFTy LinearOperators. Also our data field \( d \) is available as NIFTy Field.

Since \( S \) is only diagonal in the harmonic space the sampling itself takes place in that representation. Another NIFTy LinearOperator defined as \( HT \) transforms fields from the harmonic to the position representation. The energy for the full posterior in equation (27) is already implemented as Energy in NIFTy, too.

With that an HMCSampler instance can be defined as:

```python
import nifty4 as ift
import hmcf
[...]
posterior_energy = ift.library.WienerFilterEnergy(position=start_position, d=d, R=R, N=N, S=S, inverter=inverter)
wiener_hmc = hmcf.HMCSampler(potential=posterior_energy, num_processes=5, sample_transform=HT)
```

where `start_position` is just a field in the harmonic representation of the space for \( s \) with zero value everywhere. The `inverter` is a tool in NIFTy for numerically inverting operators.

To optimize the sampling process additional features are set:
wiener_hmc.epsilon = hmcf.EpsilonPowerLaw
wiener_hmc.mass.reevaluations = 2
wiener_hmc.display = hmcf.TableDisplay

And finally a run is initiated:

```python
wiener_hmc.run(num_samples=200, convergence_tolerance=0.5)
```

**Results**

Afterwards the HMC mean is compared to the analytic solution of the problem:

```python
hmc_mean = wiener_hmc.mean
analytic_mean = HT(D*R.adjoint*N.inverse(d))
diff = abs(hmc_mean - analytic_mean)
```

and visualized using the **NIFTy** plot functions:

```python
lo = hmc_mean.min()
hi = hmc_mean.max()
plotdict = {"colormap": "Planck-like", "ymin": lo, "ymax": hi}
ift.plot(hmc_mean, name="hmc_mean.png", **plotdict)
ift.plot(analytic_mean, name="analytic_mean.png", **plotdict)
ift.plot(diff, name='difference.png', **plotdict)
```

When running the script the HDF5 file is saved to a (new) subdirectory “samples” where the script is located.

Additionally the last lines in the code above generate plots using the Python package **matplotlib** in the directory from where the script was started.

Figure 3 displays these plots generated using `numpy.random.seed(42)`.

### 5.2. Log Normal

The Wiener filter example is nice as an proof of concept but does not exactly show the advantages of an HMC approach. The analytical solution calculated via conjugate gradients is essentially as fast as one leapfrog integration step in the HMC approach and gives a nearly perfect solution.

To present a problem where an HMC approach is actually better than other approaches we introduce a still rather simple log-normal Wiener filter scheme. Since there is no analytical solution for calculating the mean value most algorithms calculate a maximum-a-posteriori (MAP) solution which is most of the time good enough but does not in general represent the actual mean value of the distribution.

In this example, again a Gaussian prior is set on $s$

$$P(s|S) \propto \exp\left(-\frac{1}{2}s^T S^{-1}s\right)$$  \hspace{1cm} (29)
Figure 3: Results of the wiener_filter_demo.py script. The first row represents the two approaches in solving the problem, where a) is the mean of the HMC samples and b) the conjugate gradient solution. In the bottom row the absolute difference of both approaches is displayed in c) and finally in d) the mock data field from which the reconstruction was performed.

but the data model for $d$ is changed to

$$d = Re^x + n$$

with $n$ being Gaussian noise. This imposes a log-normal prior on the variable $x = e^s$ implicitly. The energy functional for the full posterior can be written as

$$H(s|d) = \frac{1}{2}s^\top S^{-1}s + \frac{1}{2}(d - Re^s)^\top N^{-1}(d - Re^s) + \text{const}$$

In this example we work on a (one-dimensional) 1024 pixel regular space to keep the sampling time relatively short.

Once again the covariance $S$ is assumed to be diagonal in the harmonic space of $s$ with its power spectrum given by (28).

Implementation in HMCF

The following code snippets can also be found in the HMCF package’s demo folder in the
nonlinear_wiener_filter_demo.py script. A similar demo is also available in NIFTy solving the problem with a MAP algorithm. In the HMCF demo we use the MAP solution to compare our mean solution to.

The demo works by default on a (one-dimensional) 1024 pixel regular space. For simplicity we assume again that for the following example $D$, $S$, $R$ and $N$ are already well defined as NIFTy LinearOperators. But our instrument $R$ is broken in that it has a blind spot 200 pixels wide where it only returns zeros to make the problem a little more interesting.

As in the case of the Wiener filter example a NIFTy LinearOperator defined as $HT$ transforms fields from the harmonic to the position representation. We further introduce a field $\tau = \text{diag}(S)^{\frac{1}{2}}$ which imprints the power spectrum in the harmonic representation of $s$ onto a white noise field. With that we can introduce a new random variable $z = \tau^{-1} \odot s$ ($\odot$ represents the Hadamard Product) for sampling which has the identity as prior covariance since

$$s^\top S^{-1} s = (\tau^{-1} \odot s)^\top (\tau^{-1} \odot s) = z^\top z.$$  \hspace{1cm} (32)

The field representation of $z$ is the prior eigenbasis.

Sampling in the space of $z$ but being interested in $e^s$ also means that we need a non-linear transformation of our samples before expectation values can be calculated:

```
import nifty4 as ift
import hmcf

[...]
non_linearity = ift.library.nonlinearities.Exponential()
[...]
def sample_transform(z):
    return non_linearity(HT(power*z))
```

The energy for the full posterior in equation (27) is already implemented as Energy in NIFTy. First we use the MAP approach of NIFTy for comparison:

```
m = ift.Field.full(h_space, 1e-7)
posterior_energy = ift.library.NonlinearWienerFilterEnergy(
    m, d, R, non_linearity, HT, power, N, S, inverter=inverter)

# Minimization with chosen minimizer
posterior_energy = minimizer(posterior_energy)[0]
map_solution = posterior_energy.position
```

where minimizer is an NIFTy algorithm returning an Energy at the MAP position.

As starting points for our Markov chains we create samples from the prior covariance $S$ (still the identity) with mean $\text{map\_solution}$:

```
x_initial = [map_solution + S.draw_sample() for _ in range(num_processes)]
```

Creating an HMCSampler instance and adjusting some parameters works as before with the Wiener filter:
\[
\text{nl_hmc} = \text{hmcf.HMCSampler}(\text{potential}=\text{posterior\_energy}, \\
\quad \text{num\_processes}=\text{num\_processes}, \\
\quad \text{sample\_transform}=\text{sample\_transform})
\]
\[
\text{nl_hmc.display} = \text{hmcf.TableDisplay} \\
\text{nl_hmc.epsilon} = \text{hmcf.EpsilonPowerLawDivergence} \\
\text{nl_hmc.mass.reevaluations} = 3 \\
\text{nl_hmc.run}(\text{num\_samples}=1000, \text{max\_burn\_in}=5000, \\
\quad \text{convergence\_tolerance}=1., \\
\quad \text{x\_initial}={x\_initial})
\]

**Results**

Afterwards the HMC mean is compared to the MAP solution of the problem:
\[
\begin{align*}
\text{hmc\_mean} & = \text{nl\_hmc.mean} \\
\text{hmc\_std} & = \text{ift.sqrt(nl\_hmc.var)} \\
\text{map\_solution} & = \text{sample\_transform(map\_solution)} \\
\text{diff} & = \text{abs(map\_solution - hmc\_mean)}
\end{align*}
\]

and again visualized:
\[
\begin{align*}
\text{lo} & = \text{np.min([true\_sky.min(), map\_solution.min(), data.min()])} \\
\text{hi} & = \text{np.max([true\_sky.max(), map\_solution.max(), data.max()])} \\
\text{plotdict} & = \{"\text{colormap": } \text{"Planck-like"}, \text{"ymin": } \text{lo}, \text{"ymax": } \text{hi}\} \\
\text{ift.plot(true\_sky, name="true\_sky.png", **plotdict)} \\
\text{ift.plot(map\_solution, name="reconstructed\_sky.png", **plotdict)} \\
\text{ift.plot(data, name="data.png", **plotdict)} \\
\text{ift.plot(hmc\_mean, name='hmc\_mean.png', **plotdict)} \\
\text{ift.plot(diff, name='difference.png', **plotdict)} \\
\text{ift.plot(hmc\_std, name='hmc\_std.png', colormap="Planck-like",} \\
\quad \text{ymin}={0.}, \text{ymax}={hmc\_var.max()})
\end{align*}
\]

These images generated with `numpy.random.seed(42)` are depicted in figure 4.

### 6. Summary

Efficient HMC sampling with the high number of degrees of freedom of a numerically represented field is a very complicated task. **HMCF** takes care of most challenges arising while working on such problems. It provides good default values and adjusting strategies for crucial parameters such as the integration step size or the mass matrix. Nonetheless the user is still able to customize many details of how the sampler deals with a given problem.

Apart from a diverse set of different options to choose from the structure of the module even eases the creation of new, customized options. We explained the usage of **HMCF** and demonstrated its performance using the demonstrator coming with the **HMCF** package.

### Acknowledgments

We would like to thank all members of the Information Field Theory group at MPA for many
Figure 4: Results of the log_normal_demo.py script. It shows that the MAP solution (figure d)) is in this case a very good approximation to the true (HMC) mean (figure c)). The HMC mean was calculated using in total 5000 samples. The problem includes a partially 'broken' instrument which can be easily seen as a prominent zero-valued feature in the data field (figure b)). In figure e) the high uncertainty of the HMC sampler in this region can be observed.
useful discussions in particular, and alphabetical order: Philipp Arras, Jakob Knollmüller, Daniel Pumpe, Reimar Leike and Martin Reinecke.
References

Betancourt M (2017). “A conceptual introduction to Hamiltonian Monte Carlo.” *arXiv preprint arXiv:1701.02434.*

Buscombe D (2016). “Spatially explicit spectral analysis of point clouds and geospatial data.” *Computers & Geosciences, 86*, 92–108.

Dalcín L, Paz R, Storti M (2005). “MPI for Python.” *Journal of Parallel and Distributed Computing, 65*(9), 1108–1115. ISSN 0743-7315. doi:https://doi.org/10.1016/j.jpdc.2005.03.010. URL http://www.sciencedirect.com/science/article/pii/S0743731505000560.

Duane S, Kennedy AD, Pendleton BJ, Roweth D (1987). “Hybrid monte carlo.” *Physics letters B, 195*(2), 216–222.

Enßlin TA, Frommert M, Kitaura FS (2009). “Information field theory for cosmological perturbation reconstruction and nonlinear signal analysis.” *Physical Review D, 80*(10), 105005.

Enßlin TA, Weig C (2010). “Inference with minimal Gibbs free energy in information field theory.” *Physical Review E, 82*(5), 051112.

Gelman A, Rubin DB (1992). “Inference from iterative simulation using multiple sequences.” *Statistical science*, pp. 457–472.

Hanson KM (2001). “Markov Chain Monte Carlo posterior sampling with the Hamiltonian method.” In *Medical Imaging 2001: Image Processing*, volume 4322, pp. 456–468. International Society for Optics and Photonics.

Hastings WK (1970). “Monte Carlo sampling methods using Markov chains and their applications.” *Biometrika, 57*(1), 97–109.

Hoffman MD, Gelman A (2014). “The No-U-turn sampler: adaptively setting path lengths in Hamiltonian Monte Carlo.” *Journal of Machine Learning Research, 15*(1), 1593–1623.

Hunter JD (2007). “Matplotlib: A 2D Graphics Environment.” *Computing in Science Engineering, 9*(3), 90–95. ISSN 1521-9615. doi:10.1109/MCSE.2007.55.

Junklewitz H, Bell M, Selig M, Enßlin T (2016). “RESOLVE: A new algorithm for aperture synthesis imaging of extended emission in radio astronomy.” *Astronomy & Astrophysics, 586*, A76.

Knollmüller J, Steininger T, Enßlin TA (2017). “Inference of signals with unknown correlation structure from non-linear measurements.” *arXiv preprint arXiv:1711.02955.*

Metropolis N, Rosenbluth AW, Rosenbluth MN, Teller AH, Teller E (1953). “Equation of state calculations by fast computing machines.” *The journal of chemical physics, 21*(6), 1087–1092.

Neal RM, *et al.* (2011). “MCMC using Hamiltonian dynamics.” *Handbook of Markov Chain Monte Carlo, 2*(11).
Oliphant TE (2007). “Python for Scientific Computing.” *Computing in Science Engineering, 9*(3), 10–20. ISSN 1521-9615. doi:10.1109/MCSE.2007.58.

Pumpe D, Reinecke M, Enßlin TA (2018). “Denoising, Deconvolving and Decomposing multi-Dimensional Photon Observations-The D4PO Algorithm.” *arXiv preprint arXiv:1802.02013*.

Reinecke M (2018). *nifty4: a package implementing numerical methods for Information Field Theory*. Max-Planck-Society. Python package version 4.1.0, URL http://ift.pages.mpcdf.de/NIFTy.

Salvatier J, Wiecki T, Fonnesbeck C (2018). *pyMC3: Python package for all sorts of MCMC algorithms*. Python package version 3.4.1, URL https://docs.pymc.io.

Selig M, Vacca V, Oppermann N, Enßlin TA (2015). “The denoised, deconvolved, and decomposed Fermi $\gamma$-ray sky-An application of the D3PO algorithm.” *Astronomy & Astrophysics, 581*, A126.

Stan Development Team (2017). *PyStan: Python package for HMC and NUTS*. Python package version 2.16.0.0, URL http://mc-stan.org.

Steininger T, Dixit J, Frank P, Greiner M, Hutschenreuter S, Knollmüller J, Leike R, Porqueres N, Pumpe D, Reinecke M, et al. (2017). “NIFTy 3-Numerical Information Field Theory-A Python framework for multicomponent signal inference on HPC clusters.” *arXiv preprint arXiv:1708.01073*.

Taylor J, Ashdown M, Hobson M (2008). “Fast optimal CMB power spectrum estimation with Hamiltonian sampling.” *Monthly Notices of the Royal Astronomical Society, 389*(3), 1284–1292.
Affiliation:

Christoph Lienhard
Max Planck Institut for Astrophysics
Karl-Schwarzschild-Str. 1
D-85741 Garching, Germany
and
Ludwig-Maximilians-Universität München
Geschwister-Scholl-Platz 1
80539 Munich, Germany
and
Technische Universität München
Arcisstr. 21
80333 Munich, Germany
E-mail: christoph.lienhard@tum.de

Torsten Enßlin
Max Planck Institut for Astrophysics
Karl-Schwarzschild-Str. 1
D-85741 Garching, Germany
and
Ludwig-Maximilians-Universität München
Geschwister-Scholl-Platz 1
80539 Munich, Germany