The No-U-Turn Sampler with dual averaging

Seminararbeit Baysianische Inferenzmethoden WS 20/21

Henri Funk

07. February 2021

Abstract

This seminar handover intends to impart the intuition and give visual understanding of the No-U-Turn sampler. Therefore, the evolution from random jumping distribution over Hamiltonian Monte Carlo to the No-U-Turn sampler is briefly described in chapter 1. Chapter 2 continues introducing the No-U-Turn sampler with dual averaging step by step. The explanation is guided by R code that implements and builds features of the sampler. Examples vizualize typical behavioral patterns of the algorithms. In Chapter 3 the implemented code is used in a regression and a classification context. The results are evaluated in typical convergence diagnostic methods. Those methods give deep insight in advantages and disadvantages of the proposed algorithms. Finally, those algorithms are evaluated for their usage and a outview on further methods and improvement is given.

Contents

List of Tables

List of Figures

List of abbreviations

Term	Abbreviation
(unormalized) negative log posterior	$L(\cdot)$
(unormalized) negative log posterior gradient	$\Delta_\theta L$
(unormalized) posterior	$p(\cdot)$
acceptance probability of iteration m	α_m
Effective Sample Size	ESS
Hamiltonian Monte Carlo	HMC
identity matrix	I
Iteration	m (index)
Iteration in Warm Up Phase	w (index)
jumping distribution	$J(\cdot)$
Leapfrog steps (amount)	$\mathcal{L}$
Markov Chain Monte Carlo	MCMC
Metropolis-Hastings	MH
momentum	r
No-U Turn Sampler	NUTS
per iteration states (quantity)	$\mathcal{B}$
per iteration valid states(quantity)	$\mathcal{C}$
position	θ
slice variable	u
state	(θ_t, r_t)
state leftmost	(θ_l, r_l)
state rightmost	(θ_r, r_r)
step size	ϵ
Time	t (index)
unnormalized joint density	$p(\theta, r) \propto exp\{L(\theta) - \frac{1}{2} < r, r >\}$

1 Introduction

The No-U-Turn sampler (NUTS) was introduced in 2014 by Hoffman and Gelman in the paper "The No-U-Turn sampler: Adaptively setting path lengths in Hamiltonian Monte Carlo". This paper suggests an algorithm that builds upon Hamiltonian Monte Carlo Samplers (HMC) and can be considered as an extension.
Therefore, NUTS regulates the amount of Leapfrog steps $\mathcal{L}$ automatically, which are considered in HMC as a tuning parameter. More precisely, NUTS adapts $\mathcal{L}$ in each iteration to fit the local energy level set.
Additionally, Hoffman and Gelman propose a method called *dual averaging* that automates HMCs tuning parameter stepsize ϵ. Finally, Hoffman and Gelman provide a framework in which both methods can be combined in a MCMC framework.
This first chapter gives a brief overview on the evolution from simple Metropolis algorithm over Hamiltonian Monte Carlo to the No-U-Turn sampler. Therefore, the focus of this chapter will be on the problem of simpler algorithms and the solutions of those problems that lead to the idea of NUTS.

1.1 Problems of random walk

Metropolis(-Hastings) Algorithm is the base family of many complex MCMC procedures. Its iterative behavior can be summarized in two main steps: *random walk* and *acceptance/rejection*.
The random walk step proposes a new position θ_{prop} based on the position θ_m of the previous MCMC step. In the second step this proposals' likelihood is compared to the likelihood from the previous iteration. Their joint likelihood ratio gives the algorithm a score between $(0, 1]$ which is used as acceptance rate for the proposal θ_{prop} to be valid for iteration m (in this case $\theta_{prop} = \theta_{m+1}$).
As θ_{prop} is chosen randomly from a *jumping distribution* $J(\theta_{prop}|\theta_m)$, the whole algorithms' convergence speed depends on $J(\cdot)$. Therefore, Gelman proposes the following criteria for $J(\cdot)$ to be a useful jumping distribution (see Gelman et al. (2013)):

- For any tuple (θ_a, θ_b) it is easy to sample from $J(\theta_a|\theta_b)$
- $J(\cdot)$ is easy to compute (e.g. if the distribution is symmetric)
- The jump has a *reasonable* distance
- We don't reject/ accept jumps too often

Despite all good effort and knowledge, slight changes in $J(\cdot)$ might harm the speed of convergence significantly. In Figure 1, Metropolis random walk behavior of 200 iterations is shown in two experimental settings. The data used are taken from *"Open Data LMU"* where we took the first two parameters of a perception experiment ("Akustische Und Perzeptive Auswirkungen Des Lächelns Auf Sprache." 2010). Both settings use a multivariate normal distribution as jumping distribution $J(\cdot)$ and identical set-up for all other tuning parameters. In the left plot, the variance-covariance estimate from generalized linear model is used while the other plot uses the variance-covariance of a simple linear model.

The left plot seems to have proper distance between jumps, where visible autocorrelation is disappearing after the first iterations. In contrast to that, jumps in the right plot seem heavily autocorrelated as the algorithm is producing too cautious, small steps.

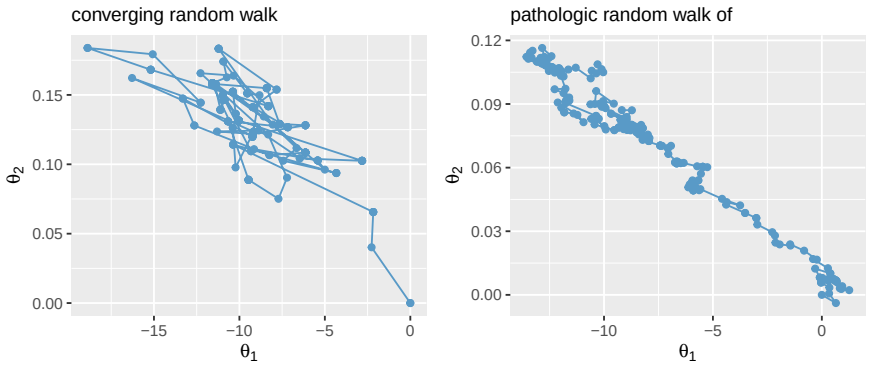

Figure 1: *Metropolis Hastings random walk sequence*

Even well chosen jumping distributions may spend a lot of time 'zigging and zagging' in the target distribution. This is a result to $J(\cdot)$ still selecting θ_{prop} at random and not including any other information than θ_t to determine the following position.

1.2 Hamiltonian Monte Carlo

The Hamiltonian Monte Carlo sampler is an algorithm that tackles the problem of random walk by equipping the jumping distribution $J(\cdot)$ with more information about the posteriors' curvature at the current position $p(\theta_m)$. It does so by introducing a momentum r_0 for each component of the parameter θ. The resulting algorithm is somewhat a *hybrid* Monte Carlo with a mix of random walk and deterministic simulation methods derived from hamiltonian dynamics.

1.2.1 Intuition - A ball in a bowl

To get a intuition of what HMC does in an iteration, one may think of an easy two dimensional experiment where a ball is dropped into a bowl (see Figure 2). The ball starts with potential but no kinetic energy. As the ball falls, potential energy is replaced by kinetic energy until the ball reaches the bottom of the bowl. Once the ball starts ascending the opposite side of the bowl, kinetic energy is traded back to potential energy and the process will repeat in the opposite direction. In a frictionless setting, this ball will descend and ascend the bowl until eternity (see Moore (n.d.)). This intuition can easily be adapted to higher dimensions.

The trajectory of the ball in the bowl over time is described as hamiltonian dynamic. Note that the height where the ball was dropped initially can be understood as the *energy level set* of the balls' trajectory. If one drops the ball lower in the bowl, the ball won't ascend higher than it was dropped and thus have a lower energy level set.

Imagine, one would catch the ball in the bowl at a random point of its trajectory and drop it from this point again and repeat this for many times. In this setting, the trajectory would get smaller

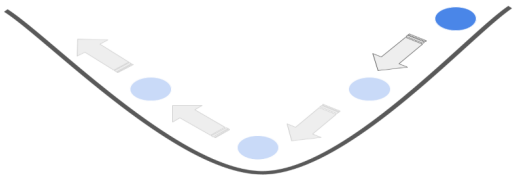

Figure 2: *Ball in Bowl experiment*

and smaller as the ball would swing closer around the bottom of the bowl.

1.2.2 Momentum - Introducing potential energy

Hamiltonian Monte Carlo exploits the relationship between potential and kinetic energy. One may think of the bowl as a posterior that needs to be explored and of the initial position where the ball was dropped as the latest MCMC sample θ_{m-1}. Note that for any posterior density function, the negative log transformation is convex and thus looks more or less like the bowl in the example (see left plot in Figure 3).

Let the trajectory of the ball through the bowl describe the energy level set $\mathcal{B}$ of all possible new positions for iteration m. This implies for the proposal of iteration m θ_{prop} to be $\in \mathcal{B}$. In case of the ball in the bowl metaphor, this implies that any position of the balls' trajectory will be at least as close to the bottom of the bowl as the position where the ball was dropped. For HMC, this means that any position $\theta_{prop} \in \mathcal{B}$ will be at least as close to the minimum as θ_{m-1}. Continuing this sampling process, $\lim_{n\to\infty} \theta_n$ would converge against the minimum of the negative log posterior.

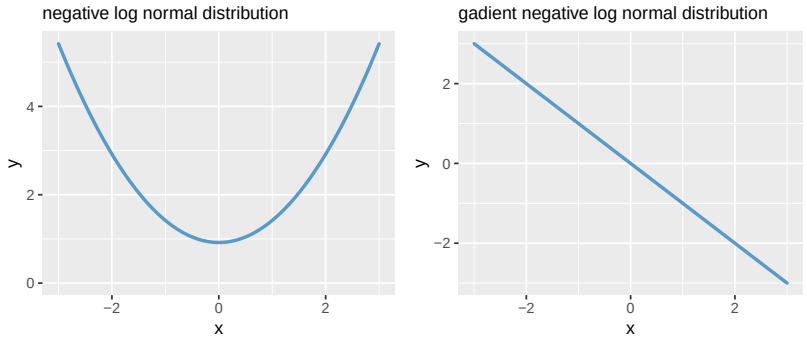

Figure 3: *Negative logarithmic Normal*

The kinetic energy can be understood as a momentum that drives the ball towards a certain direction. In Hamiltonian Monte Carlo, this momentum r_0 is largely determined by the gradient of the (unnormalized) negative log posterior density function $L(\cdot)$. In the right plot of Figure 3, the gradient of the negative log standard normal distribution $\Delta_\theta L$ is plotted exemplary. Note that the value of the gradient at a given point $\Delta L(\theta)$ gives a clear hint in which direction the minimum is to be found and how far this minimum is from the actual position. This value of the gradient is passed to the momentum, encouraging the position to take lager steps towards the minimum if it is far

away or traveling cautiously if it was dropped close to the minimum.
Note that the trajectory of the ball within the energy level set differs from the trajectory in the negative log posterior! Tracing the trajectory in $L(\theta)$ implies only tracing the ball in the bowl. The energy level set is tracing the tuple of (θ, r) in their joint density $p(\theta, r)$, so it is tracing the balls' potential and its kinetic energy, which is stable. Their unnormalized joint density is given by $p(\theta, r) \propto exp\{L(\theta) - \frac{1}{2} < r, r >\}$.

1.2.3 Leapfrog Integrator

In very few cases, Hamiltonian dynamics can be evaluated continuously. The Leapfrog Integrator provides a symplectic Integrator, that discretizes the trajectory of position in the energy level set during time t. This integration of HMC is comparable to the jumping distribution of Metropolis(-Hastings).
The R code below shows a possible implementation for a *two-stage leapfrog* integrator. The first argument describes a current `position` of the parameter in posterior space at a given time (t): θ_t. The `momentum` r_t is the kinetic energy of this position in posterior space. `Stepsize` indicates the influence of the `momentum` on the `position`. In other words this parameter specifies how ambitious or cautious the Leapfrog step is.
In a two-stage Leapfrog step the momentum is updated twice by adding $\frac{\epsilon}{2}\Delta_\theta L(\theta_t)$ to the current momentum r_t. Between those momentum updates, the position gets updated by $\theta_t + \epsilon r_t$ (see ("Choice of Symplectic Integrator in Hamiltonian Monte Carlo," n.d.)).

```
leapfrog <- function(position, momentum, stepsize) {
  momentum <- momentum + (stepsize / 2) * gradient(position)
  position <- position + stepsize * momentum
  momentum <- momentum + (stepsize / 2) * gradient(position)
  return(list("position" = as.numeric(position), "momentum" = as.numeric(momentum)))
}
```

Repeating these Leapfrog steps for a given amount of times $\mathcal{L}$, will return a new state $(\theta_{prop}, r_{prop})$ within the trajectory in the *energy level set*.

1.2.4 HMC- Code

Understanding how hamiltonian dynamics and their discretization in Leapfrog steps work is only one step away from understanding Hamiltonian Monte Carlo. The commented R code below presents a implementation option for the whole algorithm. The function has four arguments.
`position_init` is the initial position where the algorithm should start iterating. `stepsize` ϵ is passed to the `leapfrog` function. `Leapfrog steps` $\mathcal{L}$ is the amount of Leapfrog steps that HMC will take per iteration for proposing a new position θ_{prop}. `iterations` M is the amount of MCMC iterations to sample.
Note that for this implementation of HMC, it is obligatory to specify the (unnormalized) log posterior L as well as its log gradient $\Delta_\theta L = (\frac{\delta L(\theta)}{\delta_1}, ..., \frac{\delta L(\theta)}{\delta_d})$. As sufficient gradient approximations may be

costly in terms of computational effort, an analytic approximation of $\Delta_\theta L$ is necessary to preserve HMC convergence speed.

```
hamiltonianMC <- function(position_init, stepsize, leapfrogsteps, iterations) {
  position <- position_init
  positions <- data.frame(matrix(ncol = length(position_init), nrow = iterations))
  # our proposal for Leapfrog Steps
  for(iter in seq_len(iterations)) {
    momentum <- rnorm(length(position_init))
    proposal <- list("position" = position, "momentum" = momentum)
    for(step in seq_len(leapfrogsteps)){
      proposal <- leapfrog(proposal$position, proposal$momentum, stepsize)
    }
    acceptance <- exp(joint_log_density(proposal$position, proposal$momentum) -
                      joint_log_density(position, momentum))
    if(is.na(acceptance)) acceptance <- 1
    acceptance <- min(1, acceptance)
    position <- if(runif(1) < acceptance) proposal$position else position
    positions[iter, ] <- position
    setTxtProgressBar(pb, iter)
  }
  return(positions)
}
```

Like MH algorithm, HMC constitutes from two components per iteration: a partly determined, partly random walk and an acceptance-rejection step. The algorithm begins to sample a momentum $r_0 \sim N_d(0, I)$ in the same shape as the position ($\theta_m \in \mathbb{R}^d$). This initial momentum has the purpose to shift the state (θ_m, r_0) to a new energy level (see Figure 4) where the algorithm takes $\mathcal{L}$ Leapfrog steps. It applies a certain arbitrariness to the algorithm to obtain random components in the walk. Recall the ball in a bowl metaphor for intuition but think of a ball in a 3 dimensional bowl. This step would be analog to catch the ball in the bowl and then release the ball again by pushing it into a new random direction r_0. This enables the ball to explore new areas of the bowl each time it is caught and released again. Still, gravity will pull the ball towards the bottom of the bowl.
Like the jumping distribution $J(\cdot)$ in MH, iterating over Leapfrog Integrator provides a new proposal $(\theta_{prop}, r_{prop})$ at its end. This proposals' likelihood in joint distribution $p(\theta_{prop}, r_{prop})$ is compared to $p(\theta_m, r_m)$ from the previous iteration. The ratio is used analog to the ratio in MH to build an acceptance-rejection criteria for $(\theta_{prop}, r_{prop})$. It reinsures that the Leapfrog Integrator did not cause too much error by discretizing the Hamiltonian dynamic.

1.2.5 Tuning Parameter

In general Hamiltonian Monte Carlo has three important tuning Parameters:

- Mass Matrix $\mathcal{M}$
- amount of Leapfrog steps $\mathcal{L}$

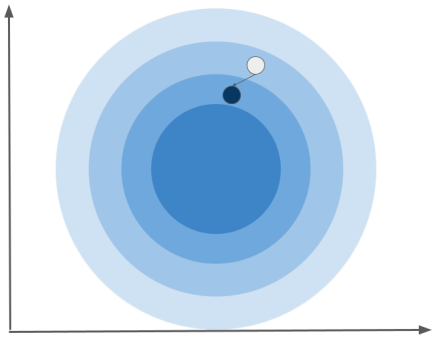

Figure 4: *Switch in energy level set - (θ_m, r_m) gets randomly shifted to a new energy level set (θ_m, r_0) in the joint posterior space*

- stepsize ϵ

Note that $\mathcal{M}$ is expected to be given as the identity matrix I in the sampled momentum $r \sim N_d(0, I)$ in the R Code above and in the following. This tuning parameter opens a whole new area of research which is tackled, e.g., by the *Riemannian adaption* but won't be part of this study.
ϵ regulates the distance each Leapfrog step takes, while $\mathcal{L}$ determines how many steps should be taken to generate a new proposal.

1.2.5.1 Desired behavior and adoptivity To take another step towards the No-U-Turn sampler, one may think of what would be a desirable behavior for ϵ and $\mathcal{L}$.

1) The amount of Leapfrog steps $\mathcal{L}$ should drive the trajectory of steps in one iteration through the whole posterior space. This would also imply that $\mathcal{L}$ is adaptive to the locally chosen energy set in each iteration. *This challenge is tackled by NUTS.*
2) A well fitting stepsize ϵ should get smaller in areas of high curvature, carefully exploiting those areas without jumping out of the energy level set. ϵ should get higher in areas with simple curvature not wasting too much computational effort. *This challenge is tackled by dual averaging.*

Both approaches may be combined. In Chapter 2 we will discuss the implementation for a No-U-Turn sampler with dual averaging, as introduced in Gelman and Hoffman (2014).

1.2.5.2 Tuning problems In some situations HMC might fail to converge because $\mathcal{L}$ and ϵ where too ambitious. Particularly, this can happen if the distribution has a complex curvature and/or is high dimensional with strong dependencies between the variables.
In these cases, hand-tuning the parameters might have high computational costs and does not necessarily lead to convergence within a feasible amount of iterations. This is when the No-U Turn Sampler (with dual averaging) becomes a valid option.

2 No-U-Turn-Sampler

NUTS is a MCMC framework that builds upon Hamiltonian Dynamics and uses, as HMC, the Leapfrog Integrator to discretize the trajectory. The main advantage of No-U-Turn compared to Hamiltonian Monte Carlo is that it provides a mechanism to automate HMCs' tuning parameter $\mathcal{L}$, the amount of Leapfrog steps.
The proposed algorithm adapts $\mathcal{L}$ in each iteration to the present energy level set of (θ_m, r_0). This secures samples from autocorrelation if $\mathcal{L}$ was set too low. Additionally, it avoids superfluous computational calculations, if $\mathcal{L}$ was set too high (Hoffman and Gelman (2014)). The automation of the tuning parameter $\mathcal{L}$ is a powerful tool, especially in high dimensional correlated data, where there is no one-value-solution to $\mathcal{L}$, but rather the need to adapt $\mathcal{L}$ to different energy level sets.
In the following chapter, a solution of the naive NUTS is presented with the purpose of giving a general understanding of the algorithms' key steps. This chapter is followed by presenting the efficient NUTS, which is faster than naive NUTS, while preserving the same functionality as naive NUTS.

2.1 Naive No-U-Turn Sampler

2.1.1 Problems with tuning $\mathcal{L}$ in HMC

Figure 5 shows the evolution of Leapfrog steps $\mathcal{L}$ in 2-dimensional posterior space $(\theta \sim N_2(0, I))$ over time.

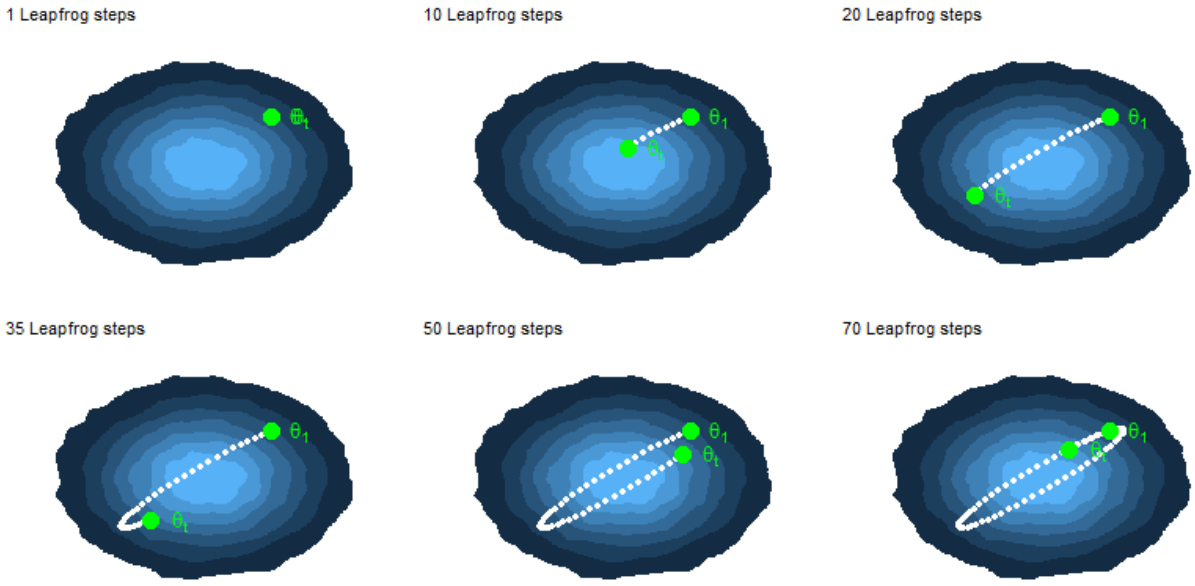

Figure 5: *Evolution of Leapfrog steps for increasing number of $\mathcal{L}$ in one HMC iteration in posterior space $N_2(0, I)(\epsilon = 0.1)$*

Choosing a lower amount of Leapfrog steps would cause autocorrelation between iteration θ_1 and

the position after taking t Leapfrog steps θ_t. Choosing a higher amount of Leapfrog steps would cause superfluous computational costs.
Note that the *perfect* amount of Leapfrog steps is not constant but rather varying between each new energy level set per iteration. This implies that a HMC algorithm with, e.g., 1000 iterations adapts to $\mathcal{L}$ in every iteration in order to fit the trajectory length to the local energy level set of the current iteration. To apply such an adaptive algorithm, we need to introduce a procedure that checks Leapfrog steps while expanding the trajectory in each iteration.

2.1.2 Automizing $\mathcal{L}$

Starting with the intuition of Hamiltonian Monte Carlo, there needs to be an idea when the trajectory should stop integrating and how to generalize this idea to universally usage.
In general, a good jumping distribution should be able to *fly* through the whole posterior space. Therefore, recall Figure 5. While the trajectories of $t = 1$ and 10 steps are clearly too close to the starting value, in case of $t = 50$ and 70 steps, the trajectory "bites its own tail" by rerunning the same trail again. Both settings represent pathological behavior that should be avoided when automating $\mathcal{L}$. In case of taking too less steps, θ_1 and θ_t will be highly autocorrelated. In case of taking too many steps, superfluous computational costs and, eventually, autocorrelated samples will decrease the algorithms' convergence speed.
The trajectories *sweet spot* can be found somewhere between $t = 20$ and $t = 30$ in the example. The attentive observers might have noticed that the trajectory drawn by the Leapfrog Integrator made a *U-Turn* in between those two steps. This behavior is exactly what gives the algorithm its name.

2.1.2.1 Criteria Having this visual criteria when to stop taking further steps, the next stage is to bring this criteria to a mathematical solution. This solution should of course be easy to apply while the algorithm is running.
In figure 6, a vector is spanned between position θ_1 and θ_t and the momentum vector r_t is spanned beginning from θ_t. Comparing these two vectors in both plots, the angle between $(\theta_t - \theta_1)$ and r_t turns from $> 90°$ in the left plot where $t = 20$ to $< 90°$ where $t = 35$.
In fact, the point in the trajectory, θopt, where the angle between the two vectors equals $90°$ is equivalent to the point where the trajectory travelled as far as possible in the energy level set. After reaching θopt, the trajectory is starting to back down towards initial θ_1. Hence, $(\theta_1 - \theta opt)$ represents the maximal distance within the regarding trajectory. This implies that the stopping criteria for Leapfrog Integration should be met at the point where the vector of $(\theta_1 - \theta_t)$ stands orthogonal on the vector of momentum r_t. The stopping criteria can be defined as

$$(\theta_1 - \theta_t) r_t \geq 0.$$

Note that this can, and should be checked in the reverse direction, too:

$$(\theta_1 - \theta_t) r_1 \geq 0$$

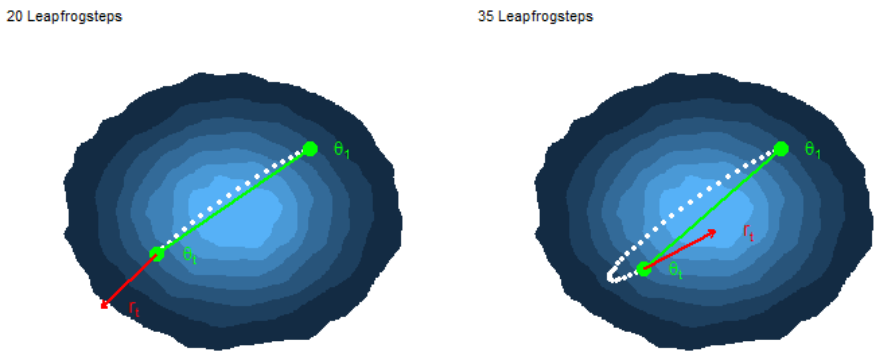

Figure 6: *Vector that spans between initial position θ_m and position after taking a given amount of t Leapfog steps θ_t and the mometum vector r_t at t-th iteration spanning from θ_t for $t = 20$ and $t = 35$*

2.1.3 Successive exploration scheme

Having defined the *U-Turn criteria*, an algorithm needs to be set around the Leapfrog steps $1, ..., t$ that is able to apply the criteria and stop the trajectory once the criteria is met. Note that is is not sufficient to compare only the initial θ_1 to the latest sampled position θ_t as the U-Turn criteria might be also triggered, e.g., between θ_t and θ_2. Therefore it is crucial for any exploration scheme, to apply the criteria among the states $(\theta_0, r_0), (\theta_1, r_1), ..., (\theta_t, r_t)$.
However, there exist various ideas of how such a scheme might look like. Anyways, the general structure of an exploration scheme looks as follows:

1) Build a trajectory with a given length
2) Check the U-Turn criterion within the sample of positions gathered from Leapfrog Integrator
3) Expand the trajectory and and repeat checks
4) Return the sample once the criterion is met

2.1.3.1 Naive additive scheme This scheme is the most basic idea and proposes to step to a new state (θ_t, r_t) and compare this state to all other sampled states $(\theta_0, r_0), (\theta_1, r_1), ..., (\theta_{t-1}, r_{t-1})$. The problematic within this idea lies in the fact that for the t-th iteration t^2 checks on the U-Turn criteria would be necessary. For higher amount of Leapfrog steps this operation might become costly and at some point unfeasible.

2.1.3.2 Multiplicative doubling scheme This scheme presents an efficient alternative to the additive scheme. Here, the amount of Leapfrog step gets iteratively doubled building a balanced binary tree (see figure 7 from Hoffman and Gelman (2014)). The U-turn criteria is then only applied within one subtree and between the other trees. Each leaf of a tree represents a state $(\theta_i, r_i), i \in 1, .., t$. This procedure results in checking the U-Turn criteria $log(t)$ times.
Note that doubling is done in both directions randomly here to remain the trajectories reversibility in time. For more details see Betancourt (2017).

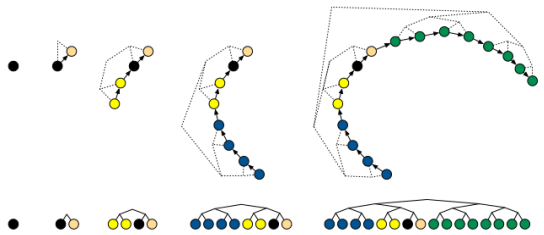

Figure 7: *Typical doubling proccedure. The initial point is black. Each colour desricbes a new subtree.*

Table 1: *Implementation of states variable*

Slot	Definition
valid_state	list containing valid positions and momenti sampled so far
rightmost	list containing rightmost state (θ_r, r_r)
leftmost	list containing rightmost state (θ_l, r_l)
run	logical indicating if the stopping criteria was met or not $(TRUE)$

2.1.4 Doubling

The implementation of such a *balanced binary tree* requires an algorithm that constantly doubles the amount of states (θ, r) until the U-Turn criteria is met. The code below shows an easy sketch of how such a algorithm could look at main level: The algorithm starts initializing `tree_depth` an indicator for the length of the trajectories expansion.
Function `build_tree` doubles the amount of states building a new proposal. `is_U_turn` checks if the U-Turn criteria is met in between the proposal and the existing state. `states$run` denotes if the u-turn criteria was met somewhere within any two subtrees. If not, the algorithm has to expand the trajectory once again. We gather all sates $\{(\theta_m, r_0), \ldots, (\theta_t, r_t)\}$ sampled so far in `states` and increase the `tree_depth`. This is done until the U-Turn criteria is met.

```
tree_depth = 0L
while(states$run){
  states_prop <- build_tree(tree_depth, ...)
  states$run <- states$run && states_prop$run
  if(states$run) states <- unite_valid_states(states, states_prop)
  tree_depth = tree_depth + 1
}
```

`states` is a key variable to this code that contains all necessary information about sampled states during doubling in one MCMC iteration of NUTS. Table 1 presents the different slots of `states`. Note that `rightmost` and `leftmost` are important information to the algorithm that identify the end points in a current trajectory. These information are crucial to indicate where to set the next leapfrog step.

A possible approach to double states is *recursion*. This procedure builds the core element of the No-U-Turn sampler.

2.1.4.1 Recursion In the following the recursion step will be bundled to a function that is called `build_tree`.

```
build_tree <- function(tree_depth = d, ...) {
  if(tree_depth == 0) {
    states <- build_leaf(...)
  } else {
    states <- build_tree(tree_depth = d - 1L, ...)
    states_prop <- build_tree(tree_depth = d - 1L, ...)
    states$run <- is_U_turn(states, states_prop)
    state <- unite_valid_states(states, states_prop)
  }
  return(state)
}
```

In its base case `build_tree` builds one "leaf". A leaf of a tree represents a state with position θ and momentum r. `build_leaf` functionality will be introduced in the next section. Until here, it is sufficient to know that this function produces a Leapfrog step and returns it to `build_tree`. For simplicity, assume all other arguments are given, which is summarized by a ellipsis ... in the sketch code.

The amount of Leapfrog steps is determined by the recursions depth `tree_depth`. Any call of `build_tree` with a given tree depth d will induce 2^d Leapfrog steps.

This `build_tree` function calls itself twice, reducing the depth parameter d by 1 at each recursion step. In Figure 8, a recursion in `build_tree` is shown exemplary for `tree_depth = 3`.

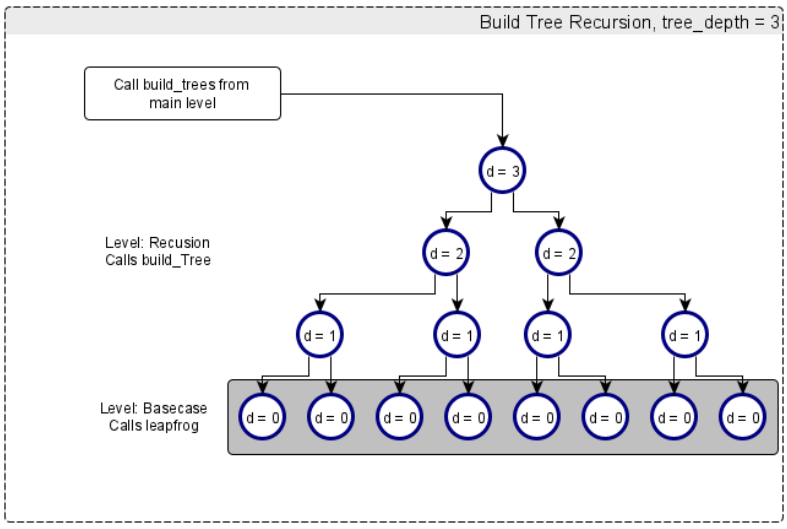

Figure 8: *Example recursion procedure at a tree depth of 3*

Called from a main level, this function will double itself in each depth resulting in a set of $2^3 = 8$ Leapfrog steps. Each of those steps produces a state $state_i = (\theta_i, r_i), i \in \{1, \ldots, 8\}$. At each level leftmost and rightmost states (θ_l, θ_r) will get checked by the U-Turn criteria.

- At base level the algorithm checks u-turn criteria pairs of
$$(state_1, state_2), (state_3, state_4), (state_5, state_6), (state_7, state_8)$$
- At level $d = 1$ checks are done for pairs of
$$(state_1, state_4), (state_5, state_8)$$
- At level $d = 2$ U-Turn criterion is checked for pair of
$$(state_1, state_8)$$

2.1.4.2 Direction In HMC Leapfrog steps implicitly move in one direction. This direction is determined by the gradient and represents the shortest way to the minimum of the log posterior $L(\theta)$. Unlike HMC, the doubling process in NUTS must be able to expand the trajectory in the energy level set in both directions randomly. To implement this feature of doubling fore- and backwards in time, we simply introduce a variable `direction` $\sim Unif(\{-1, 1\})$ that is called once in each doubling. In essence this variable gets passed down to `leapfrog` function calling it with a directional hint.

```
leapfrog(position, momentum, direction * stepsize)
```

2.1.4.3 Bidirectional Doubling In the following, a naive bidirectional doubling procedure is described. Therefore, `build_leaf` is set up as a primitive function, that calls `leapfrog` function and initializes a state with position and momentum coordinates from this Leapfrog step. `build_tree` is extended in a way that the function is able to deal with different directions. Therefore, it does not only need to pass information down to `build_leaf`, but also deal with different starting positions for Leapfrog integration.
Function `initialize_states` creates a `state`, while `is_U_turn(states)` checks whether the U-turn criteria is met between `state$rightmost` and `state$leftmost`. The code for `initialize_state` and `is_U_turn` can be found in the appendix.

```
build_leaf <- function(position_momentum, direction, stepsize) {
  step <- leapfrog(position_momentum$position, position_momentum$momentum,
                   stepsize = (direction * stepsize))
  state <- initialize_states(step$position, step$momentum)
  state$valid_state <- step
  state
}

build_tree <- function(position_momentum, direction, tree_depth, stepsize, ...) {
  if(tree_depth == 0L) {
    build_leaf(position_momentum, direction, stepsize)
    } else {
      states <- build_tree(position_momentum, direction, tree_depth - 1L, stepsize)
```

```
    if(direction == -1L) {
      states_prop <- build_tree(states$leftmost, direction, tree_depth - 1L, stepsize)
      position_momentum  <- states$leftmost <- states_prop$leftmost
    } else {
      states_prop <- build_tree(states$rightmost, direction, tree_depth - 1L, stepsize)
      position_momentum  <- states$rightmost <- states_prop$rightmost
    }

    states$run <- states_prop$run * is_U_turn(states)
    states$valid_state$position <- rbind(states$valid_state$position,
                                         states_prop$valid_state$position)
    states$valid_state$momentum <- rbind(states$valid_state$momentum,
                                         states_prop$valid_state$momentum)
    return(states)
  }
}
```

For the bidirectional doubling procedure, we need to initialize a gradient $\Delta_\theta L$. In this example we are given $\theta \sim N_2(0, I)$. This implies $\Delta_{\theta_i} L = -\theta_i$. We set stepsize $\epsilon = 0.1$ and initialize $tree\_depth = 0$. Before initializing the proposal, we randomly sample a direction for the doubling procedure. Note that the algorithm needs to update the tree on its surface once again. This update of two `states` is analog to the update in `build_tree`.

```
states_sample <- list()
set.seed(123L)
gradient <- function(x) -x
stepsize <- 0.1
states <- initialize_states(position = c(1, 1), momentum = rnorm(2))
tree_depth <- 0L
while(states$run){
  direction <- sample(c(-1, 1), 1)
  if(direction == -1L) {
    states_prop <- build_tree(states$leftmost, direction, tree_depth, stepsize)
    states$leftmost <- states_prop$leftmost
    } else {
    states_prop <- build_tree(states$rightmost, direction, tree_depth, stepsize)
    states$rightmost <- states_prop$rightmost
    }
  states$run <- is_U_turn(states) * states$run * states_prop$run
  if(states$run) {
    states$valid_state$position <- rbind(states$valid_state$position,
                                         states_prop$valid_state$position)
    states$valid_state$momentum <- rbind(states$valid_state$momentum,
                                         states_prop$valid_state$momentum)
  }
```

```
  tree_depth = tree_depth + 1
  states_sample[[tree_depth]] <- states_prop$valid_state$position
}
```

In figure 9 the results from the bidirectional algorithm are plotted for each doubling. The green colored Leapfrog steps indicate doubling where the U-Turn criteria wasn't met, while the red colored steps indicate the last doubling where the U-Turn criteria was met. The algorithm is stops visibly once the U-Turn is taken by Leapfrog steps in their trajectory.

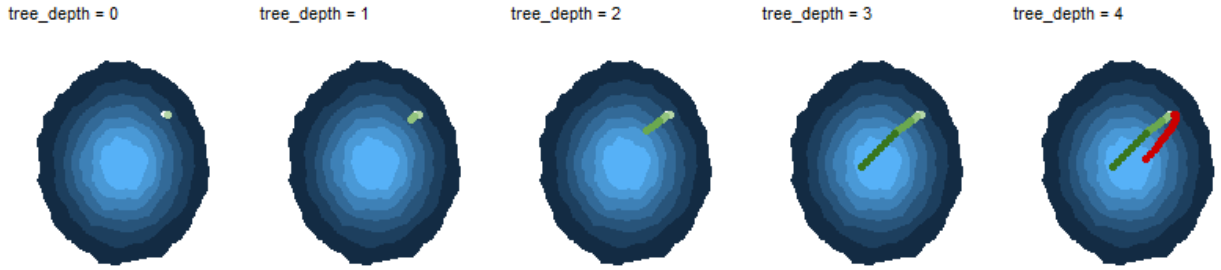

Figure 9: *Evolution of Leapfrog steps ($\epsilon = 0.1$) for bidirectional doubling in $N_2(0, I)$. The green dots show doubling procedures where U-Turn critria is not met. The red dots show Leapfrog steps where U-Turn criteria was met. The algorithm is doubling backwards, than three times foreward and in the last doubling backwards.*

The presented algorithm provides a component that is comparable to the jumping distribution in Metropolis Hastings. The output of this function retrieved in this while loop, is the set $\mathcal{B}$ which contains all states from Leapfrog integration, but those the last doubling iteration. The next chapter present a framework to control discretization of the Leapfrog Integrator.

2.1.5 Pathologic Behavior

Recall set $\mathcal{B}$ that constitutes of all states sampled from Leapfrog integration until the U-Turn criteria was met. If Leapfrog integration would have been perfect, all of these states $(\theta_i, r_i) \in \mathcal{B}$ would have the same joint probability $p(\theta_i, r_i)$ as they where sampled from the same energy level set. Unfortunately Leapfrog discretization might lead to some errors from time to time (see e.g. ("Choice of Symplectic Integrator in Hamiltonian Monte Carlo," n.d.)). Especially, if ϵ was set to high discretization steps might jump out of the energy level set into low posterior density regions. These state samples have to be condemn because they exhibit pathological behavior.

2.1.5.1 The slice sampling check Slice sampling provides an alternative to check the sampled states for pathologic behavior while preserving the invariance of MCMC. For more information about the math behind this process see Hoffman and Gelman (2014).
To filter states that exhibit pathologic behavior, we introduce a *slice variable* $u \sim Unif(0, p(\theta_m, r_0))$. While building up the trajectory, u serves as threshold for the likelihood of states received from the Leapfrog integrator.

Therefore, we need to implement the joint density of position and momentum $p(\theta, r)$ to the inventory of helper functions:

```
log_posterior_density <- function(position) log(mvtnorm::dmvnorm(position))

joint_log_density <- function(position, momentum) {
  log_dens_estimate <- log_posterior_density(position)
  log_dens_estimate - 0.5 * sum(momentum * momentum)
}
```

The code is a straight forward implementation of the proposed joint density $p(\theta, r)$ from Hoffman and Gelman (2014):

$$p(\theta, r) \propto exp\{L(\theta) - 0.5 < r, r >\}$$

where $< \cdot, \cdot >$ is the dot product of any vector $\cdot$.
Note that for any distribution we must also implement a function `log_posterior_density(position)` that is the implementation of an (unnormalized) logarithmic density estimate for the posterior and that takes a position vector with all positions that shall be sampled during NUTS iterations.
For more detailed elaboration of slice sampling see Neal (2003).

2.1.5.2 Valid states By filtering exactly those states $(\theta_i, r_i), i \in 1, \ldots, t$ from $\mathcal{B}$ that satisfy $p(\theta_i, r_i) \geq u$, we implicitly create a new set $\mathcal{C} \subseteq \mathcal{B}$. $\mathcal{C}$ contains exactly those states $(\theta_i, r_i), i \in 1, \ldots, t$ that satisfy the slice check $p(\theta_i, r_i) \geq u$.

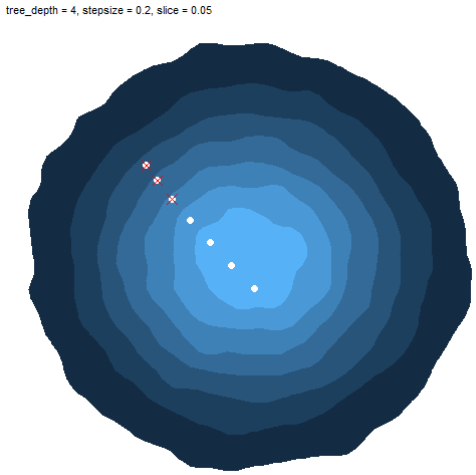

Figure 10: *Evolution of Leapfrog steps ($\epsilon = 0.1$) for bidirectional doubling in $N_2(0, I)$. The red crosses show the points that will be discraded because they exhibit pathologic behavior.*

In Figure 10, Leapfrog steps within a trajectory are plotted (white). The red crossed steps exhibit Leapfrog steps that are discarded by slice sampling. Note that the slice sampler will be more likely to discard points that tend to drive the trajectory in low density regions of the posterior L_θ as only those steps are discarded that have a low joint density $p(\theta, r)$.

2.1.5.3 One additional stopping criteria

The slice can also be applied to add an additional stopping criteria to the sampling process. If discretization error from Leapfrog steps is way to high the whole trajectory process should be stopped. This is important because the discretization error might increase from one Leapfrog step to another leading to a whole "pathologic" sample. To avoid that Hoffman and Gelman (2014) introduce Δmax, a threshold that can stop doubling when crossed.

$$\begin{aligned} L(\theta_i, r_i) - log(u) &< -\Delta_{max} \\ = log(\frac{p(\theta_i, r_i)}{u}) &< -\Delta_{max} \end{aligned}$$

Δmax compares the actual sampled state with the slice u. Typically Δmax is set high, to "not interfere with the algorithm so long as the simulation is even moderately accurate." Hoffman and Gelman (2014)

2.1.5.4 Implementation

The slice u can easily be sampled by calling the joint density at NUTS main level. Note that the u needs to be sampled after the new momentum is initialized.

```
momentum <- rnorm(length(position))
dens <- joint_log_density(position, momentum)
slice <- runif(n = 1, min = 0, max = exp(dens))
```

This slice it then passed all the way down to `build_leaf` function. Here, we can make `states$valid_state` selective. Leapfrog step (θ_i, r_i) will only be in the set of valid states $\mathcal{C}$ if $p(\theta_i, r_i) \geq u$.
For the Δ_{max} stopping criteria we compare $p(\theta_i, r_i)$ with the slice u. Δ_{max} is set to 1000 here.

```
build_leaf <- function(position_momentum, direction, stepsize, slice) {
  step <- leapfrog(position_momentum$position, position_momentum$momentum,
                   stepsize = (direction * stepsize))
  state <- initialize_states(step$position, step$momentum)
  dens <- joint_log_density(step$position, step$momentum)
  if(slice <= exp(dens)) state$valid_state <- step
  state$run <- (dens - log(slice)) >= -1e3
  state
}
```

2.1.6 Transition kernel

Until here, the provided framework is able to return a set $\mathcal{C}$ that contains all `valid_states` within the trajectory of an energy level set in one MCMC iteration. We must now define a transition kernel that returns a sample for the next iteration θ_{m+1} while leaving the distribution over $\mathcal{C}$ invariant. Hoffman and Gelman (2014) prove that this can be done by sampling one position θ_i randomly from $\mathcal{C}$.

```
position <- states$valid_state$position[sample(nrow(states$valid_state$position), 1),]
```

A code for naive NUTS can be found in the appendix.

2.2 The efficient No-U-Turn Sampler

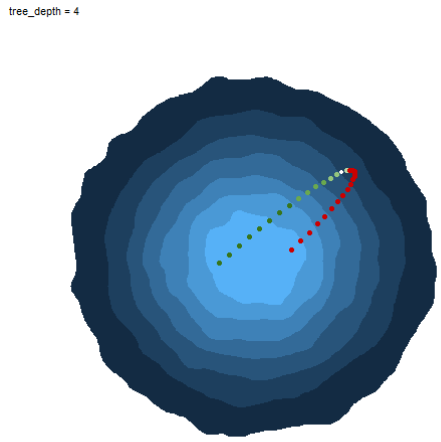

Figure 11: *Evolution of Leapfrog steps ($\epsilon = 0.1$) for bidirectional doubling in $N_2(0, I)$. The red points will be discraded because as u-turn was made during this doubling process.*

Recall the example from previous chapter in Figure 11. The Sampler does the U-Turn in the 5-th doubling sampling 2^4 new Leapfrog steps. This last doubling step should be critically examined. In the plot, the U-Turn is made already after the first Leapfrog steps of the final doubling. However, the algorithm will continue naively doubling until all 16 Leapfrog steps are done. Especially, in high dimensions and at high tree depth this doubling is costly given the fact that all of those steps from the last doubling will be discarded anyways.
Efficient NUTS is an extension to naive NUTS using a more sophisticated transition kernel. This transition kernel will be applied in the baseline functions `buils_tree` and `build_leaf`. On this level we can enable the algorithm to stop the trajectory within any subtree $\mathcal{C}_{sub}$ of $\mathcal{C}$ while maintaining the invariance over $\mathcal{C}$ (see Hoffman and Gelman (2014)).

2.2.1 Implementation

To do so, we must implement a counter `states$count` into our states variable at base level `build_leaf`. The counter counts all leapfrog steps that satisfy the slice check. Additionally, a step is only then assigned to the set of valid states $\mathcal{C}$ if this step satisfies the slice check.

```
build_leaf <- function(position_momentum, direction, stepsize, slice) {
  step <- leapfrog(position_momentum$position, position_momentum$momentum,
                   stepsize = (direction * stepsize))
  state <- initialize_states(step$position, step$momentum)
  dens <- joint_log_density(step$position, step$momentum)
```

```
  if(slice <= exp(dens)) {
    state$valid_state <- step
    state$count = 1L
  }
  state$run <- (dens - log(slice)) >= -1e3
  state
}
```

At level of `build_tree`, we sum up the counter, indicating how many valid states where drawn so far in each subtree $\mathcal{C}_{sub}$. That enables the algorithm to *update* the valid state, rather than storing all possible valid states in $\mathcal{C}$.

```
build_tree <- function(position_momentum, direction, tree_depth, stepsize, slice){
  if(tree_depth == 0L) {
    build_leaf(position_momentum, direction, stepsize, slice)
    } else {
      states <- build_tree(position_momentum, direction, tree_depth - 1L,
                           stepsize, slice)
      if(states$count) { #<<
        if(direction == -1L) {
          states_prop <- build_tree(states$leftmost, direction,
                                    tree_depth - 1L, stepsize, slice)
          position_momentum  <- states$leftmost <- states_prop$leftmost
          } else {
            states_prop <- build_tree(states$rightmost, direction,
                                      tree_depth - 1L, stepsize, slice)
            position_momentum  <- states$rightmost <- states_prop$rightmost
          }
        tree_ratio <- states_prop$count / (states_prop$count+states_prop$count)#<<
        if(rbinom(1, 1, tree_ratio)) {
          states$valid_state <- states_prop$valid_state #<<
        }
        states$count <- states_prop$count + states_prop$count #<<
        states$run <- states_prop$run * is_U_turn(states)
        }
      return(states)
    }
  }
```

Note that the NUTS is now able to exit from doubling within the recursion process. This reduces the computational effort significantly and increases the amount of Leapfrog steps in $\mathcal{C}$ as the trajectory can get closer to the U-Turn. It also minimizes memory as only one valid state has to be stored.

2.2.2 Summary

NUTS can be summarized to the following 6 steps (see Nishio and Arakawa (2019)):

1) Set the initial value of θ, ϵ and decide for a proper amount of `iterations` for the algorithm to converge.

In each iteration **DO**:

2) Generate momentum r_0 from the standard normal distribution $r \sim N(0, I)$.
3) Generate slice variable u from the uniform distribution $u \sim Uniform(0, p(\theta, r))$.
4) Generate valid states $\mathcal{C}$ by using the doubling method and sample a proposals $(\theta_{prop}, r_{prop})$ from $\mathcal{C}$ iteratively.
5) Accept the proposal $(\theta_{prop}, r_{prop})$ with probability proportional to the valid set size.
6) Repeat steps 2 to 5 until convergence.

The code for efficient NUTS can be found in the appendix.

2.3 Automating stepsize ϵ

Step size ϵ is the parameter that regulates the distance that each Leapfrog step takes during an HMC or NUTs iteration. Well tuned ϵ might be crucial to the convergence, especially, in high dimensional posteriors with strong correlation between the parameters. In such cases, stepsize needs to be set very low (e.g $\epsilon = 1e^{-10}$). This is when computations on regular computers quickly become infeasible because the numbers get too close to zero or to $\pm\infty$. The algorithm will often throw errors (mostly NaN) for calculations with those values.

To avoid this behavior automation of stepsize is necessary. The following chapters provide a solution to this task.

2.3.1 Intuition - Acceptance probability

To automate the stepsize, it is crucial to define a criteria that is able to judge if ϵ was set correctly in the algorithm. Recall, that a stepsize estimate that has a good fit to explore a (unnormalized) posterior should be low enough to cause no error while discretizing the Hamiltonian dynamic but high enough to waste no computational effort.

A good estimate in Hamiltonian Monte Carlo, to judge if ϵ was set wisely, is the acceptance probability of an iteration m : α_m.

$$\alpha_m = min\left\{\frac{p(\theta_{prop}, r_{prop})}{p(\theta_{m-1}, r_{m-1})}, 1\right\}$$

α_m constitutes from the ratio of the (unnormalized) joint density of at the proposal state $p(\theta_{prop}, r_{prop})$ versus the density of the state that was accepted in the previous iteration $p(\theta_{m-1}, r_{m-1})$. Let the average acceptance rate be defined as $\alpha = \frac{1}{M}\sum_{m=1}^{M} \alpha_m$.

Gelman et al. (2013) claims that the *acceptance sweet spot* is at 65% average acceptance rate α. This does not apply for any posterior but is a good estimate by empirical evidence. In fact, if $\alpha < 0.65$ leapfrog jumps are too ambitious, rejecting too many proposals. On the other hand, if $\alpha > 0.65$ leapfrog jumps are too cautious.

No matter if 65% is perfect or not α serves well as estimate for the performance of HMC and NUTS

Table 2: *Implementation of states variable*

Slot	Definition
valid_state	list containing valid positions and momenti sampled so far
rightmost	list containing rightmost state (θ_r, r_r)
leftmost	list containing rightmost state (θ_l, r_l)
run	logical indicating if the stopping criteria was met or not ($TRUE$)
count	counter for valid states in $\mathcal{C}$ (only active in efficient NUTS)
acceptance	reproduced sum of acceptance probabilities of Leapfrog steps
steps	counter for Leapfrog steps

w.r.t. ϵ.

As there is no acceptance rejection step in NUTS, Hoffman and Gelman (2014) reproduce this step in `build_tree` and `build_leaf` functions. This step is adapted by calculating the average acceptance over all Leapfrog steps until the U-Turn criterion is met in an iteration m.

$$\alpha_m = \frac{1}{T}\sum_{t=1}^{T} min\left\{\frac{p(\theta_t, r_t)}{p(\theta_m, r_m)}, 1\right\}$$

An implementation of this algorithm in R is provided by the functions `build_tree_da` and `build_leaf_da` in the appendix. Those functions will basicly gather information about *sum of ratio* $\sum_{t=1}^{T} min\left\{\frac{p(\theta_t, r_t)}{p(\theta_m, r_m)}, 1\right\}$ in `state$acceptence` slot and *amount of Leapfogsteps* T in `state$steps`. See table 2 for all information within `states` variable.

2.3.2 Adaptive tuning

The idea behind adaptive tuning ϵ is very simple. Split NUTS into an warm up phase with W iterations where ϵ is dynamically adapted and a stationary phase with M iterations where ϵ_W gained from warm up phase is hold constant.

The aim of th warm up phase is to find a stepsize ϵ that guarantees quick convergence and set this value in stationary phase to make the algorithm converge. Therefore suppose we aim for a target average acceptance probability $\delta(= 0.65)$ and the average acceptance at iteration w is α_w.

2.3.3 Dual Averaging

Let $H_w = \delta - \alpha_w$ then be the MCMC behavior at iteration w. The goal is to update ϵ_w in a way that H_{w+1} gets close to zero. To reach this state update ϵ_w as follows:

$$\epsilon_w = \epsilon_{w-1} - \eta_w H_w,\ \eta_w \in (0, 1]$$

If acceptance α_w was too high we encourage the algorithm for larger jumps, rising ϵ_w. If acceptance α_w was too low we encourage the algorithm for smaller jumps, decreasing ϵ_w.
The problem is that parameters are usually quite different between warm-up and stationary phase. In optimal conditions, the stepsize would adapt quickly while exploring in warm-up phase but be set to a robust estimate in stationary phase. This is why **dual averaging** proposes a more sophisticated framework than the raw idea proposed above.
Dual averaging updates H_w as a weighted sum over all previous $H_j, j \in \{1, \dots, w-1\}$. This makes H_w more robust to changes for high w.

$$H_w = (1 - \frac{1}{w_0 + w})H_{w-1} + \frac{1}{w_0 + w}(\delta - \alpha_w)$$

To update stepsize, dual averaging proposes two parameters. One to use as stepsize in warm up phase. This ϵ_w is able to adapt quickly to changes in the local trajectory within the posterior:

$$log(\epsilon_w) = \mu - \frac{\sqrt{w}}{\gamma}H^w$$

And one that is *trained* during warm up for usage in stationary phase by developing a robust stepsize for any trajectory within the posterior:

$$log(\bar{\epsilon}_w) = w^{-\kappa}log(\epsilon^w) + (1 - w^{-\kappa})log(\bar{\epsilon}_{w-1})$$

All parameters in the equation have a default value that was proposed by empirical research in Hoffman and Gelman (2014). Those parameters default value, a description of what kind of behavior they inhibit and a straight forward implementation of dual averaging through the functions `init_parameters` and `update_stepsize` in R can be found in the appendix. Note that the only tuning parameter of those functions on the surface of the main sampling function will be the target acceptance δ.
Plots in figure 12 show the development of stepsize ϵ and $\bar{\epsilon}$ by time for constant target and average acceptance. Note that this is an experimental setting as average acceptance will vary over iterations. Note also that both plots use different scales, as $\epsilon \in (0, \infty)$ converges exponentially in the experiment. If target acceptance is constantly lower than average acceptance (right plot) steps are too cautious, wasting computational effort. So the algorithm will increase stepsize towards ∞. If target acceptance is constantly higher than average acceptance (left plot) errors from discretization are too high and the algorithm will shrink stepsize towards zero. The peak in early iterations of the "cautious experiment" in the left plot is caused by parameter μ that encourages the algorithm to take rather high stepsizes. However this effect is quickly vanishing. The epsilon for stationary phase adapts a bit delayed and smoothed which makes it more robust to changes in the posterior curvature.

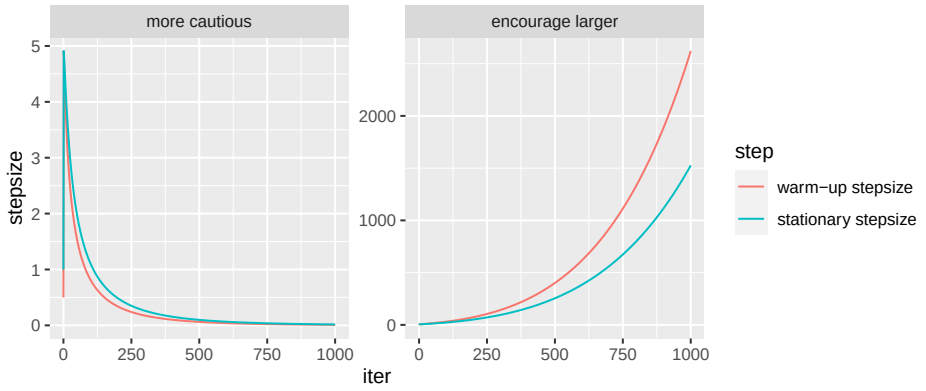

Figure 12: *development of stepsize ϵ and $\bar{\epsilon}$ by time for constant target and average acceptance. Target acceptance is 0.65 in both plots, while average accaptance is set to 0.64 in the left plot and 0.66 in the right plot.*

2.3.4 Set initial stepsize ϵ_0

To automatize ϵ completely, a mechanism needs to be implemented that is searching for a good initial step size ϵ_0. Hoffman and Gelman (2014) propose an algorithm `find_initial_stepsize` that is simulating the first Leapfrog step within an energy level set starting at the initial position. The algorithm basically retries this first step until the acceptance rate $\alpha_{init} = \frac{p(\theta_{prop}, r_{prop})}{p(\theta_{init}, r_{init})}$ is within a given range $\alpha_{init} \in [0.5, 2]$.

The algorithm initializes the stepsize $\epsilon_0 = 1$. If $\epsilon_0 = 1$ was too ambitious, the mechanism constantly halves ϵ_0 until reaching the interval. If $\epsilon_0 = 1$ was too cautious, it constantly doubles the stepsize until reaching the interval. The attached code shows a possible integration of this algorithm in R.

```
find_initial_stepsize <- function(position) {
  stepsize <- 1
  momentum <- rnorm(length(position))
  proposal <- leapfrog(position, momentum, stepsize)
  exp <- 2 * (acceptance_rate(position, momentum, proposal) > 0.5) - 1
  while(acceptance_rate(position, momentum, proposal)^exp > 2^(-exp)) {
    stepsize <- stepsize * 2^exp
    proposal <- leapfrog(position, momentum, stepsize)
  }
  stepsize
}
# ------------------------------------------------------------------------
acceptance_rate <- function(position, momentum, proposal) {
  proposal_dens <- do.call(joint_log_density, proposal)
  initial_dens <- joint_log_density(position, momentum)
  exp(proposal_dens - initial_dens)
}
```

2.4 Efficient NUTS with dual averaging

The No-U-Turn sampler with dual averaging can be summarized to the following 7 steps (see Nishio and Arakawa (2019)):

1) Set the initial value of $theta$, ϵ and decide for a proper amount of `iterations` for the algorithm to converge.

In each iteration **DO**:

2) Generate momentum r_0 from the standard normal distribution $r \sim N(0, I)$.
3) Generate slice variable u from the uniform distribution $u \sim Uniform(0, p(\theta, r))$.
4) Generate valid states $\mathcal{C}$ by using the doubling method and sample a proposals $(\theta_{prop}, r_{prop})$ from $\mathcal{C}$ iteratively.
5) Accept the proposal $(\theta_{prop}, r_{prop})$ with probability proportional to the valid set size.
6) Update ϵ_m by dual averaging.
7) Repeat steps 2 to 5 until convergence, step 6 is repeated only during the warm-up phase.

The following code provides an implementation for NUTS with dual averaging in R.

```
sample_NUT_da <- function(init_position, iteration, warmup, target_accpentance,
                          seed = 123L){
  set.seed(seed)
  pb <- txtProgressBar(min = 0, max = iteration, style = 3)
  position <- init_position
  stepsize <- find_initial_stepsize(init_position) #<
  duala_param <- init_parmeters(stepsize)
  positions <- data.frame(matrix(ncol = length(init_position), nrow = iteration))
  for(iter in seq_len(iteration)){
    momentum <- rnorm(length(position))
    dens <- joint_log_density(position, momentum)
    if(is.na(dens)) {
      warning(paste("NUTS sampled NA in iteration", iter))
      dens = 1
      }
    slice <- runif(n = 1, min = 0, max = exp(dens))
    states <- initialize_states(position, momentum)
    tree_depth <- 0L
    while(states$run){
      direction <- sample(c(-1, 1), 1)
      if(direction == -1L) {
        states_prop <- build_tree_da(states$leftmost, direction, tree_depth,
                                     stepsize, slice, dens)
        states$leftmost <- states_prop$leftmost
        } else {
          states_prop <- build_tree_da(states$rightmost, direction, tree_depth,
                                       stepsize, slice, dens)
          states$rightmost <- states_prop$rightmost
```

```
        }
      states$run <- is_U_turn(states) * states$run * states_prop$run
      if(states_prop$run) {
        tree_ratio <- min(1, states_prop$count / states$count)
        if(rbinom(1, 1, tree_ratio)) {
         states$valid_state <- states_prop$valid_state
        }
      }
      states$count <- states_prop$acceptance + states_prop$count #<<
      tree_depth = tree_depth + 1
    }
    if(iter <= warmup)  {
      average_acceptance <- states$acceptance / states_prop$step
      duala_param <- update_stepsize(duala_param, iter,
                                     target_accpentance, average_acceptance)
      stepsize <- duala_param$stepsize[iter + 1]
    }
    if(iter == (warmup + 1)) stepsize <- duala_param$stepsize_weight[iter + 1]
    setTxtProgressBar(pb, iter)
    if(is.matrix(states$valid_state$position)) {
      positions[iter,] <- position
      next
    }
    position <- as.numeric(states$valid_state$position)
    positions[iter,] <- position
  }
  return(positions)
}
```

3 Empiric and Application

This chapter shall serve as a proof of concept for the presented functions. Additionally, functionality and behavior of the discussed algorithms will be compared and evaluated. Note that the presented functions are partly supplemented with some defensive features to provide the function from errors or unfeasible run time.

3.1 Linear regression

In this chapter, a linear regression is embedded in NUTS and HMC to track convergence. The data are generated synthetically and with almost perfect linear dependency to the target. Hence, this example shall work as a proof of concept.

3.1.1 Data

The data generation process is very easy. We aim for intercept $\theta_0 = 1$ and slopes $\theta_1 = 2, \theta_2 = 3$. Note that the `target` consists of the linear dependency between `perfect_position` and `design` and noise.

```
set.seed(123L)
design <- cbind(1, sapply(1:2, function(x) runif(400)))
perfect_position <- c(1,2,3)
target <- design %*% perfect_position + rnorm(400)
```

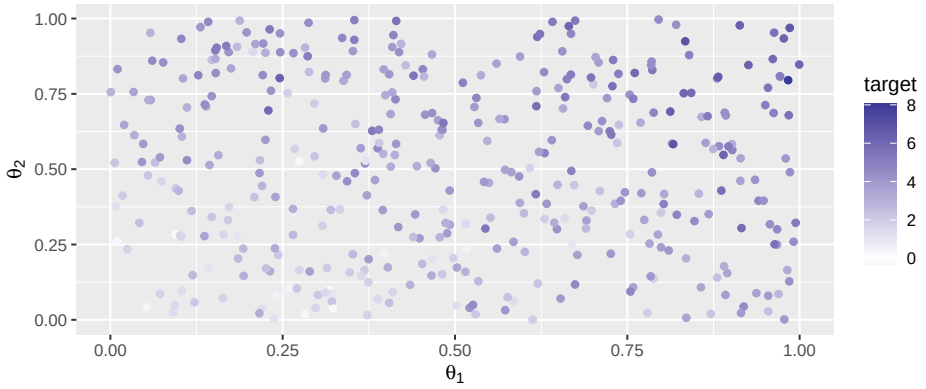

Figure 13: *Artificially generated two dimensional data with linear dependency to the target.*

3.1.2 Modelling - convergence diagnostics and tracking

To obtain a relation of the algorithms performance, Hamiltonian Monte Carlo will be compared with efficient NUTS (auto-tuning the amount of Leapfrog steps $\mathcal{L}$) and efficient NUTS with dual averaging (auto-tuning stepsize ϵ and $\mathcal{L}$).

3.1.2.1 Set Up First,, we have to define the posterior and respective gradient. The regarding derivation and code implementation can be found in the appendix. Once the functions are implemented, they need to be assigned to `log_posterior_density` and `gradient` which are called in `leapfrog` and `joint_log_density` computations.

```
log_posterior_density = log_linear
gradient = partial_deriv_lin
```

All algorithms will run for 2000 MCMC iterations and start at the same initial position `c(4,4,4)`. Hamiltonian MC and efficient NUTS use a stepsize of `0.15`. Additionally, HMC takes 8 Leapfrog steps per iteration.
NUTS with dual averaging uses the first 1000 iterations as warm-up to tune the stepsize. To set a threshold for the per iteration computation time in NUTS with dual averaging, we set the maximal tree depth `max_depth` to 13 allowing the algorithm for maximally $2^{13+1} - 1 = 16383$ Leapfrog steps

Table 3: *results from call linear modell on design and target*

	Estimate	Std. Error	t value	Pr(>|t|)
(Intercept)	0.998408	0.1327090	7.523286	0
V1	2.221141	0.1752850	12.671594	0
V2	2.792049	0.1706782	16.358554	0

per iteration. This regulates computation time significantly and makes NUTS with dual averaging usable for local and private machines.
For comparison a linear model is set up and results are shown in table 3.

```
# Hamiltonian MC
linear_sample_h <- hamiltonianMC(c(4, 4, 4), stepsize = 0.15,
                                 Leapfrog steps =  8, iterations = 2e3)
# efficient NUTS
linear_sample_l <- sample_NUT(c(4, 4, 4), stepsize = 0.15, iteration = 2e3)
# efficient NUTS with dual averaging
linear_sample <- sample_NUT_da(c(4, 4, 4), iteration = 2e3,
                               warmup = 1e3, max_depth = 13)
```

```
# Simple linear model
lm1 <- lm(target~., data = as.data.frame(design[,-1]))
```

3.1.2.2 Evaluation Trace plots from all three algorithms in Figure 14 provide a visual indication of stationarity. However, the trace plots remained for parameters from NUTS with dual averaging look a bit wiggly, indicating autocorrelation between drawn samples. This might be a sacrifice to the artificially set maximal tree depth argument which is forcing the trajectory to stop before the U-Turn was made. The parameters in Hamiltonian MC seem more stable than in NUTS with dual averaging, but clearly the most stable behavior is obtained by the MCMC iterations of efficient NUTS (middle column). The samples swing very constant around the true parameters indicated by the red line.
This assumption is confirmed by the effective sample size (calculated by `coda`) in table 4. The ESS for all samples is the highest for efficient NUTS followed by HMC and NUTS with dual averaging in that order.
Note that this is not a representative statement for NUTS with dual averaging since this algorithm was forced to quit after a given tree depth. Rather than that, one could expect NUTS to outperform the other two algorithms if the tree depth had no limit. Still computation time would increase significantly in this scenario.

3.2 Classification - Endometrial Cancer data set

Load `Endometrial` data from `hmclearn` package and standardize the data in the same way as in Thomas (2020). Data as well as posterior and gradient are described in more detail in the appendix.

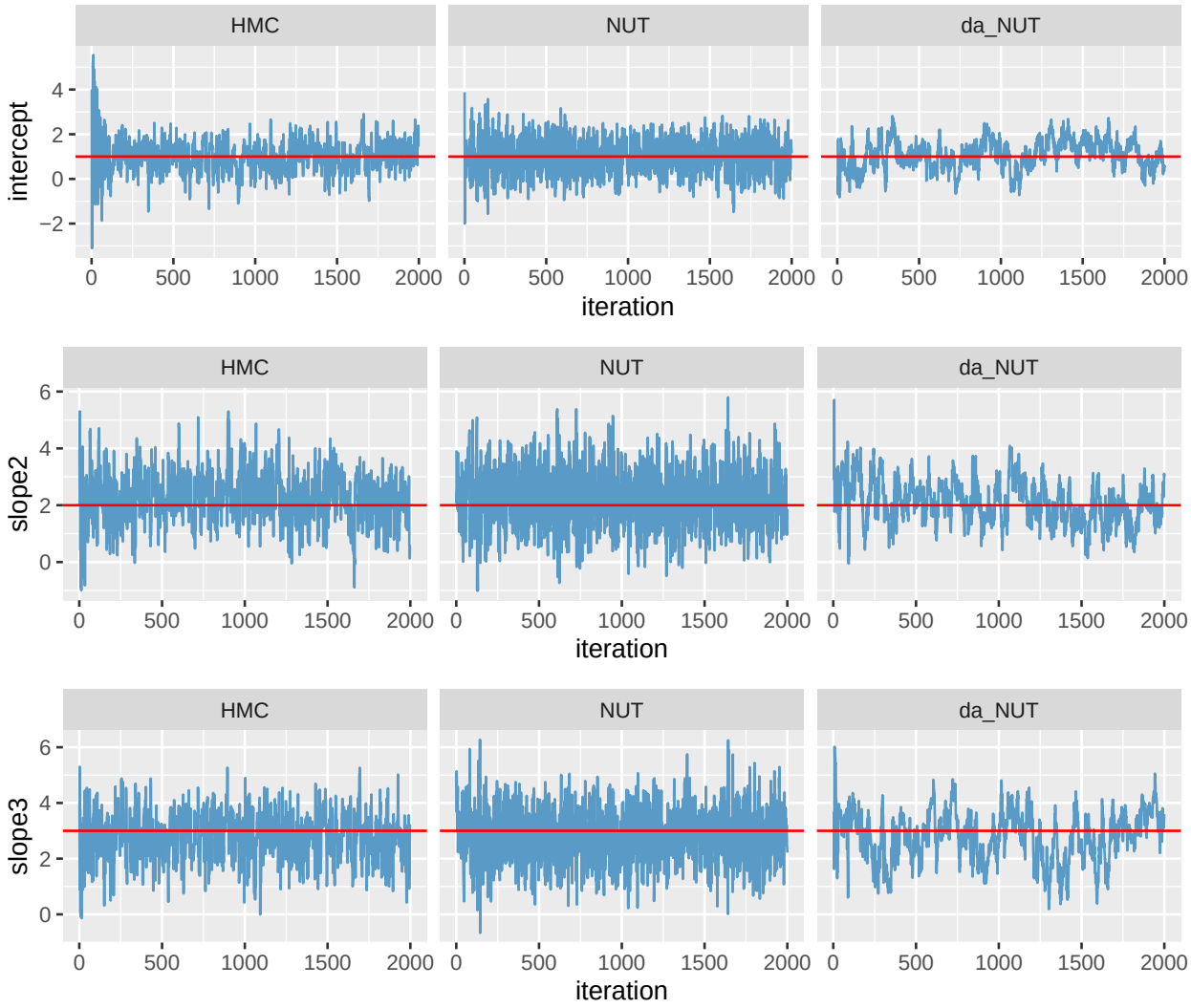

Figure 14: *Traceplots for parameters gained from linear regression gained from efficient NUTS with dual averaging, efficient NUTS and Hamiltonian MC*

Table 4: *Effective Sample Size (ESS)*

	Intercept	Slope1	Slope2
HMC	361	347	529
efficient NUTS	782	849	1014
dual averaging NUTS	50	81	49

3.2.1 Modelling - convergence diagnostics and tracking

3.2.1.1 Set up This example supposes to assign a weak penalized bernoulli posterior $L_\theta =$ `bernoulli_pen` to predict the binary target and $\Delta_\theta L =$ `partial_deriv_benoulli` as its gradient.

```
log_posterior_density = bernoulli_pen
gradient = partial_deriv_benoulli
```

To compare convergence for different starting samples θ_{init}, the algorithm is started twice from different positions $\in [-5, 5]$, while holding all other parameters constant. The first 2000 iterations will be discarded as *burn-in*. For comparison, a regularized GLM is fitted.

```
initial_pos <- lapply(1:2, function(x) runif(4, -5, 5))
sample_logistic <- lapply(initial_pos, sample_NUT, iteration = 4e3,
                          stepsize = 0.05, seed = 134)
```

```
f <- glmnet::glmnet(design, target, family = "binomial", lambda = 1/200)
```

3.2.1.2 Evaluation Figure 15 shows the trace plots for both setp-ups gained from efficient NUTS. Each models' parameter trace trough the sampling iterations is indicated by a color. The red line shows the estimate from the regularized GLM and therefore serves as comparison.
Both set ups have analog stationary traces. While `PI2` and `EH` swings very densely around the estimated value received from the penalized GLM, the intercept and `NV2` have a rather erratic convergence path. At some iterations, it seems like the samples are kicking out of the convergence path to higher positive numbers. This might result from those variables not having a significant impact on the target `HG`. As their "kicks" are only towards high positive numbers they still have a clear tendency.
The density estimates for the parameters in Figure 16 align with the assumptions gained from the trace plots. `PI2` and `EH` density estimates look rather symmetric and uni-modal. For these parameters especially `Iter 2` made a good job as the curvature looks very smoothed. As indicated by the trace plots the intercept and and slope `NV2` have a rather complex density. Both are extremely right skewed.
Fortunately it applies for each of the four parameters that the modes of their density aligns with the estimates from th regularized GLM indicated by the red line.
Table 5 shows parameter estimates dragged out from both set-ups of efficient NUTS and the regularized GLM. To get a point estimate for the comparison, the median of each parameter was evaluated (without iterations gained from burn-in). The results for Iteration 1 and Iteration 2 are rather equal and have te same tendency as the parameters from the regularized GLM.

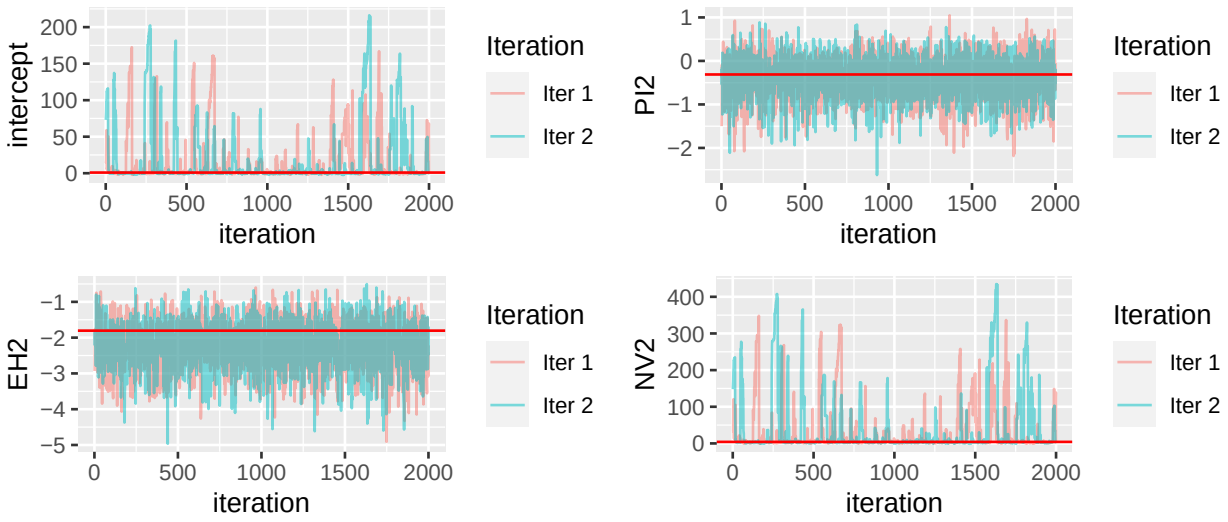

Figure 15: *Traceplots for parameters binary logistic regression gained from 2 runs of efficient NUTS.*

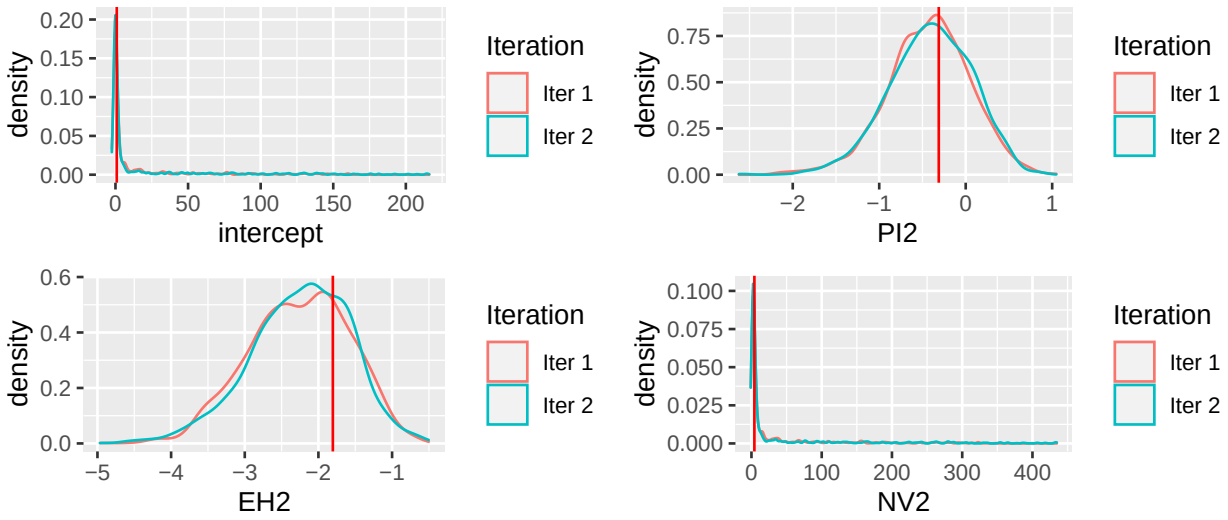

Figure 16: *Density estimates for each sampled parameter. The red line indicates the value obtained from a regularized GLM.*

Table 5: *Parameter Estimates (median in case of MCMC models)*

	Intercept	PI2	EH2	NV2
Iteration 1	0.56	-0.39	-2.21	3.69
Iteration 2	0.56	-0.39	-2.21	3.69
regularized GLM	0.87	-0.31	-1.80	4.14

4 Comparison, review and outlook

The automation of the amount of Leapfrog steps $\mathcal{L}$ and stepsize ϵ is both done for good reason. In many real world examples MCMC frameworks are applied because there is not much evidence and knowledge about the (unnormalized) posterior or the posterior is very complex and high dimensional. In such cases, it is not easy to define a fitting value for $\mathcal{L}$ and ϵ. More than that, in many cases there might be no "one-value-solution" to these parameter.

Still, the benefits of the automation of those hyperparameters come at certain costs. In many cases, the computational effort of NUTS will exceed the effort by HMC. Probably the biggest downside of these algorithms is that the costs are not predictable before starting the algorithms' run.

Comparing efficient NUTS with given stepsize and HMC with the same stepsize and $\mathcal{L}$ such that $\epsilon\mathcal{L} \approx 1$, efficient NUTS might take not more than twice the calculation time of HMC. While efficient NUTS costs are somewhat feasible, NUTS with dual averaging might cause computational costs that keep private computers running for days or even week. These high costs have their origin in dual averaging setting ϵ to very small numbers of $1e^{-10}$ or even smaller. Such small step sizes result in computing thousands and ten thousands of Leapfrog steps in each iteration. A collateral of having such small values is that numbers might become too small for, e.g., R, to really calculate with them. Unfortunately, the implemented maximal tree depth does not seem to be a good solution as it is cutting the trajectory before the criteria is met. This will in many cases cause autocorrelation between the samples. More than that, it is undermining the U-Turn criteria itself and should therefore not be applied.

Rather than forcing the trajectory to quit after a given length of Leapfrog steps, one should consider to auto-tune only $\mathcal{L}$, using efficient NUTS. Hand tune different step sizes by, e.g., black box optimization might still be a superior alternative if computation time is limited. Although, a ϵ that is set too high might cause some irregularities in the trajectory, in many cases convergence won't be harmed. Recall the parameter trace plot of the classification problem. Figure 17 shows the samples from "problematic" parameter distributions of efficient NUTS and HMC. The "kicks" in NUTS are rather intense, compared to those of HMC. Nevertheless, NUTS is able to recover from these kicks quickly while HMC suffers from them for multiple iterations. This effect is due to the fact that NUTS has not specified $\mathcal{L}$ and, therefore, is able to loop back to the true estimate within only one iteration. Analog, HMC needs hundreds of iterations to get back to the true estimates.

This is also reflected in table 6 where point estimates from HMC iteration are added. The median point estimate of the non-problematic parameters `PI2` and `EH2` is almost the same for HMC as for NUTS. But the point estimates for the intercept and `NV2` have a strong bias.

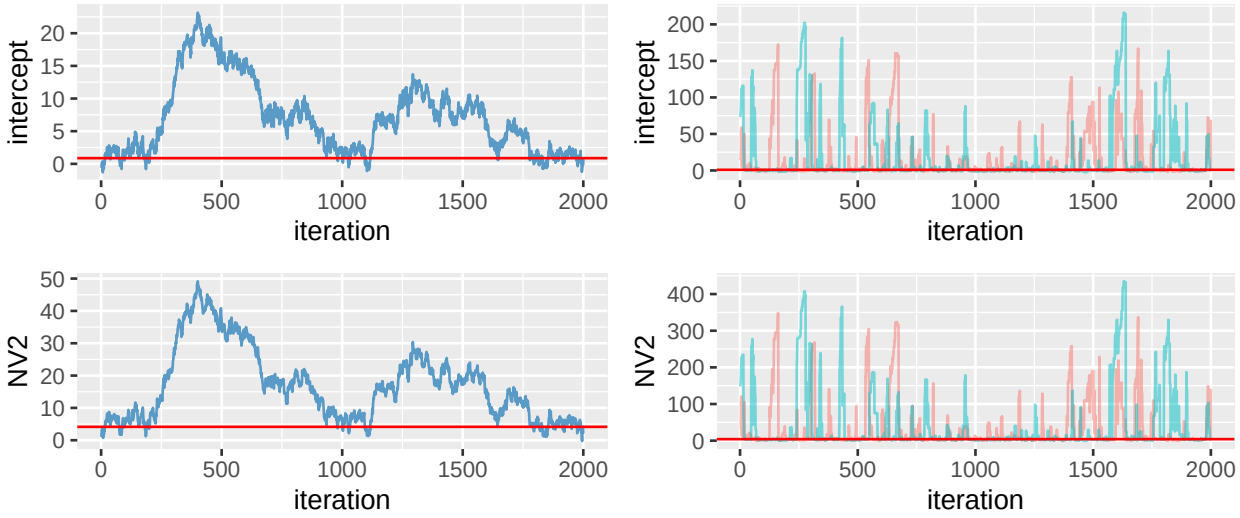

Figure 17: *Traceplots for parameters binary logistic regression gained from HMC.*

Table 6: *Parameter Estimates (median in case of MCMC models)*

	Intercept	PI2	EH2	NV2
NUTS Iter 1	0.56	-0.39	-2.21	3.69
NUTS Iter 2	0.56	-0.39	-2.21	3.69
HMC	6.45	-0.47	-2.12	15.81
regularized GLM	0.87	-0.31	-1.80	4.14

All in all the No-U-Turn sampler is a recommendable extension to HMC, especially, in cases where the samples from HMC are strongly autocorrelated. No-U-Turn with dual averaging is only then a valid alternative if neither HMC, nor NUTS converge within a feasible amount of iterations. The usage of this advanced algorithm demands a powerful calculator and sufficient time for calculation. Finally, three points remain for improvement in the provided code:

1) If momentum r_0 could be sampled with a tunable, or an adaptive mass matrix, relations between the parameters could be mapped, and thus, be implemented in the sample updates.
2) Unfortunately, R is only capable to calculate with numbers of around $2e-16$. Densities, especially in higher dimensions, might be very vast and thus have flat densities estimates. However, some packages extend this possibilities and could be implemented to the given functions to actually deal with those high dimensional densities.
3) More efficient calculation. This is a very broad and undefined approach but especially in cases of very high tree depth parallelization of doubling procedures could become a considerable option. Also base calculations and, especially, posterior and gradient functions should be set up wisely as they will be called thousands of times during one run.

5 Attachment

Naive NUTS in R

```
#_------------------------------------------------------------------------------
#' Naive No U Turn Sampler
#'
#' Sample positions with naive u turn sampler
#'
#' @param init_position vector; intial position to start MCMC
#' @param stepsize numeric; size of Leapfrog steps
#' @param iteration integer; amount of MCMC iterations
#' @param seed integer; for reproducability
#' @export
sample_nNUT <- function(init_position, stepsize, iteration, seed = 123L){
  set.seed(seed)
  pb <- txtProgressBar(min = 0, max = iteration, style = 3)
  position <- init_position
  positions <- data.frame(matrix(ncol = length(init_position), nrow = iteration))
  for(iter in seq_len(iteration)){
    momentum <- rnorm(length(position))
    dens <- joint_log_density(position, momentum)
    if(is.na(dens)) {
      warning(paste("NUTS sampled NA in iteration", iter))
      dens = 1
    }
    slice <- runif(n = 1, min = 0, max = exp(dens))
    states <- initialize_states(position, momentum)
    slice <-
      tree_depth <- 0L
    while(states$run){
      direction <- sample(c(-1, 1), 1)
      if(direction == -1L) {
        states_prop <- build_tree(states$leftmost, direction, tree_depth, stepsize)
        states$leftmost <- states_prop$leftmost
      } else {
        states_prop <- build_tree(states$rightmost, direction, tree_depth, stepsize)
        states$rightmost <- states_prop$rightmost
      }
      states$run <- is_U_turn(states) * states$run * states_prop$run
      if(states$run) {
        states$valid_state$position <- rbind(states$valid_state$position,
                                             states_prop$valid_state$position)
        states$valid_state$momentum <- rbind(states$valid_state$momentum,
                                             states_prop$valid_state$momentum)
```

```
      }
      tree_depth = tree_depth + 1
    }
    setTxtProgressBar(pb, iter)
    valid_steps <- nrow(states$valid_state$position)
    if(valid_steps == 0) {
      positions[iter,] <- position
      next
    }
    position <- states$valid_state$position[sample(seq_len(valid_steps), 1),]
    positions[iter,] <- position
  }
  return(positions)
}
```

Helper Functions in R

```
#_------------------------------------------------------------------------------
#' Initialize state
#'
#' Helperfunction to intialize state
#' @inheritParams leapfrog
#' @param run is u-turn made?
initialize_states <- function(position, momentum, run = 1L) {
  list(
    "valid_state" = structure(list(matrix(,nrow = 0, ncol = length(position)),
                                   matrix(,nrow = 0, ncol = length(position))),
                              names= c("position", "momentum")),
    "rightmost"= list("position" = position, "momentum" = momentum),
    "leftmost" = list("position" = position, "momentum" = momentum),
    "run" = run, "count" = 0, "acceptance" = 0, "steps" = 1
  )
}

# _------------------------------------------------------------------------------
#' Is a U Turn made?
#'
#' Investigates if trajectory makes a U-Turn (if >= 0)
#' @param state state variable containing rightmost and leftmost position/doubling
is_U_turn <- function(state) {
  momentum_l <- state$leftmost$momentum
  momentum_r <- state$rightmost$momentum
  if(anyNA(c(state$rightmost$momentum, state$leftmost$momentum))){
    warning("momentum contains NA, trajectory aborted")
```

```
    return(FALSE)
  }
  position_distance <- state$rightmost$position - state$leftmost$position
  left <- sum(momentum_l * as.numeric(position_distance)) >= 0
  right <- sum(momentum_r * as.numeric(position_distance)) >= 0
  left * right
}
#_------------------------------------------------------------------------------
#' Joint logarithmic density
#'
#' joint log density of position and momentum
#' @inheritParams leapfrog
joint_log_density <- function(position, momentum) {
  log_dens_estimate <- log_posterior_density(position)
  joint_dens <- log_dens_estimate - 0.5 * sum(momentum * momentum)
  if(is.na(joint_dens)) {
    warning("joint density induced NAs, density is set to -Inf")
    return(-Inf)
  }
  joint_dens
}
```

Efficient NUTS in R

```
#' Sample No-u-Turn
#'
#' efficient NUTS introduced by Hoffman and Gelman
#'
#' @inheritParams  sample_nNUT
#' @export
sample_NUT <- function(init_position, stepsize, iteration, seed = 123L){
  set.seed(seed)
  pb <- txtProgressBar(min = 0, max = iteration, style = 3)
  position <- init_position
  positions <- data.frame(matrix(ncol = length(init_position), nrow = iteration))
  for(iter in seq_len(iteration)){
    momentum <- rnorm(length(position))
    dens <- joint_log_density(position, momentum)
    if(is.na(dens)) {
      warning(paste("NUTS sampled NA in iteration", iter))
      dens = 1
    }
    slice <- runif(n = 1, min = 0, max = exp(dens))
    states <- initialize_states(position, momentum)
```

```
    tree_depth <- 0L
    while(states$run){
      direction <- sample(c(-1, 1), 1)
      if(direction == -1L) {
        states_prop <- build_tree(states$leftmost, direction,
                                  tree_depth, stepsize, slice)
        states$leftmost <- states_prop$leftmost
      } else {
        states_prop <- build_tree(states$rightmost, direction,
                                  tree_depth, stepsize, slice)
        states$rightmost <- states_prop$rightmost
      }
      if(states_prop$run) {
        if(is.na(is_U_turn(states))) browser()
        states$run <- is_U_turn(states) * states_prop$run
        tree_ratio <-min(1, states_prop$count / states$count) #<<
        if(is.na(tree_ratio)) tree_ratio <- 0
        if(rbinom(1, 1, tree_ratio)) {
          states$valid_state <- states_prop$valid_state #<<
        }
      } else break
      states$count <- states_prop$count + states_prop$count #<<
      tree_depth = tree_depth + 1
    }
    setTxtProgressBar(pb, iter)
    if(is.matrix(states$valid_state$position)) {
      positions[iter,] <- position
      next
    }
    position <- as.numeric(states$valid_state$position)
    if(anyNA(position)) stop("position contains NAs")
    positions[iter,] <- position
  }
  return(positions)
}
```

Dual averaging in R

Build tree and leaf in R - embed dual averaging

```
build_tree_da <- function(position_momentum, direction, tree_depth,
                          stepsize, slice, dens_0){
  if(tree_depth == 0L) {
    build_leaf_da(position_momentum, direction, stepsize, slice, dens_0)
```

```
    } else {
      states <- build_tree_da(position_momentum, direction, tree_depth - 1L,
                              stepsize, slice, dens_0)
      if(states$count) { #<<
        if(direction == -1L) {
          states_prop <- build_tree_da(position_momentum, direction,
                                       tree_depth - 1L, stepsize, slice, dens_0)
          position_momentum  <- states$leftmost <- states_prop$leftmost
          } else {
          states_prop <- build_tree_da(position_momentum, direction,
                                       tree_depth - 1L, stepsize, slice, dens_0)
            position_momentum  <- states$rightmost <- states_prop$rightmost
          }
        tree_ratio <- states_prop$count / (states_prop$count+states_prop$count)#<<
        if(rbinom(1, 1, tree_ratio)) {
          states$valid_state <- states_prop$valid_state #<<
        }
        states$steps <- states$steps + states_prop$acceptance
        states$acceptance <- states$acceptance + states_prop$acceptance
        states$count <- states_prop$count + states_prop$count #<<
        states$run <- states_prop$run * is_U_turn(states)
        }
      return(states)
    }
}

build_leaf_da <- function(position_momentum, direction, stepsize, slice, dens_0) {
  step <- leapfrog(position_momentum$position, position_momentum$momentum,
                   stepsize = (direction * stepsize))
  state <- initialize_states(step$position, step$momentum)
  dens <- joint_log_density(step$position, step$momentum)
  if(slice <= exp(dens)) {
    state$valid_state <- step
    state$count = 1L
  }
  state$run <- (dens - log(slice)) >= -1e3
  state$acceptance <- min(1, exp(dens - dens_0))
  if(is.na(state$run)) {
    warning("Deltamax induced NaN, moving in low posterior regions?")
```

```
    state$run = 1L
  }
  state
}
```

Dual averaging parameters- default and explanation

- w is the iteration we are at in warm-up phase
- w_0 stabilizes H_w in early iterations (default: 10L)
- ϵ_w stepsize at iteration w
- μ freely choosen value where we shrink towards (default: $log(10\epsilon_1)$, which encourages the algorithm for larger ϵ)
- γ controls the amount of shrinkage (default: 0.05)
- $w^{-\kappa}$ stepsize schedual: increases the influence of more recent iterations

see Hoffman and Gelman (2014)

Dual averaging in R

```
find_initial_stepsize <- function(position) {
  stepsize <- 1
  momentum <- rnorm(length(position))
  proposal <- leapfrog(position, momentum, stepsize)
  exp <- 2 * (acceptance_rate(position, momentum, proposal) > 0.5) - 1
  while(acceptance_rate(position, momentum, proposal)^exp > 2^(-exp)) {
    stepsize <- stepsize * 2^exp
    proposal <- leapfrog(position, momentum, stepsize)
  }
  stepsize
}

acceptance_rate <- function(position, momentum, proposal) {
  proposal_dens <- do.call(joint_log_density, proposal)
  initial_dens <- joint_log_density(position, momentum)
  exp(proposal_dens - initial_dens)
}

#' Initialize parameters for dual averaging
#'
#' @param stepsize_weight weighted stepsize from previous iteration
#' @param mcmc_behavoir behavior of algorithm at previous iterations
```

```r
#' @param shrinkage controls the amount of shrinkage
#' @param stability controls stability at early iterations
#' @param adaption controls the speed of converges and so the influence of more recent weights
#' @example dap <- init_parmeters(0.25)
#'
init_parmeters <- function(stepsize, stepsize_weight = 1,
                           mcmc_behavior = 0, shrinkage = 0.05,
                           stability = 10, adaption = 0.75) {
  level <- log(10 * stepsize) # where stepsize is shrunken towards (\mhy)
  list(
    "stepsize" = stepsize,
    "level" = level, "stepsize_weight" = stepsize_weight,
    "mcmc_behavior" = mcmc_behavior, "shrinkage" = shrinkage,
    "stability" = stability, "adaption" = adaption
  )
}



#' update stepsize
#'
#' updates stpesize parameters by performing dual averaging
#'
#' @inheritParams sample_noUturn
#' @param dap dual averaging parameters (seee init_parameters)
#' @param iter actual iteration cycle
#' @param average_acceptance average achieved acceptance in iteration m
#' @param target_acceptance aimed average acceptance level per iteration
#' @example dap <- update_stepsize(dap, 1, 0.65, 0.5)
#'
update_stepsize <- function(dap, iter, target_accpentance, average_acceptance) {
  list2env(dap, environment())
  dif <- target_accpentance - average_acceptance
  weight <- (iter + stability)^(-1)
  dap$mcmc_behavior[iter + 1] <- (1 - weight) * mcmc_behavior[iter] + weight * dif
  dap$stepsize[iter + 1] <- exp(level - ((sqrt(iter) / shrinkage) * dap$mcmc_behavior[iter + 1]
  dap$stepsize_weight[iter + 1] <- exp(iter^(-adaption) * log(dap$stepsize[iter + 1]) +
                                         (1 - iter^(-adaption)) * log(stepsize_weight[iter]))
  dap
}
```

Linear Model - Derivation and Gradient

Define the linear log Likelyhood of a `target` $y_i \sim N(x_i^t\theta, \sigma^2)$ as follows:

$$L(y|X,\theta) \propto log\Big(\prod_{i=1}^{n} exp\Big\{\frac{(y - x_i^t\theta)^2}{\sigma^2}\Big\}\Big)$$
$$\propto \sum_{i=1}^{n} exp\Big\{\frac{(y - x_i^t\theta)^2}{\sigma^2}\Big\}$$

And take it as our given Likelyhood L_θ. This can be implemented straight forward in a function `log_linear`:

```
log_linear <- function(position, sigma = 10L){
  sum((-2*sigma^(-2))*(target - design %*% position)^2)
}
```

We can build the partial derivative $\Delta_\theta L$ directly from L_θ:

$$\Delta_\theta L = \Big(\frac{\delta L(\theta)}{\delta\theta_1}, \ldots, \frac{\delta L(\theta)}{\delta_d}\Big)$$
$$\propto -\frac{1}{\sigma^2} X^t \Big(- y + X\theta\Big)$$

and implement it as follows:

```
partial_deriv_lin <- function(position,sigma =10L){
  -sigma^(-2) * t(design) %*% (-target + design %*% position)
}
```

Classification Data

The data were first published in ("Foundations of Linear and Generalized Linear Models" 2015).
Goal: Predict whether a person has endometrial cancer or not.
Dataset: A data frame with 79 rows and 4 standardized variables:

- NV Neovasculation risk factor indicator (0=Absent, 1=Present)
- PI Pulsatility index of arteria uterina
- EH Endometrium height

A comparable use case is presented by Thomas (2020) in the vignettes of `hmclearn` package. Posterior and gradient are presented there and can be easily implemented in R as follows:

```
bernoulli_pen <- function(position, design = NULL, target = NULL, sigma = 1e4) {
  if(is.data.frame(position)) position <- as.numeric(position)
  comp1 <- as.numeric(t(position) %*% t(design) %*% (target - 1))
  comp2 <- sum(log(1 + exp(-(design %*% position))))
  penal <- as.numeric(((2 * sigma) ^(-1)) * t(position) %*% position)
  comp1 - comp2  - penal
```

```
}
partial_deriv_benoulli <- function(position, design = NULL, target = NULL, sigma = 1e4){
  sigmoid_exp <- exp(-(design %*% position))
  comp1 <- (target - 1) + (sigmoid_exp / (1 + sigmoid_exp))
  comp2 <- t(design) %*% comp1
  comp2 - (position / sigma)
}
```

6 References

"Akustische Und Perzeptive Auswirkungen Des Lächelns Auf Sprache." 2010. https://data.ub.uni-muenchen.de/31/.

Betancourt, Michael. 2017. "A Conceptual Introduction to Hamiltonian Monte Carlo." *arXiv Preprint arXiv:1701.02434*.

"Choice of Symplectic Integrator in Hamiltonian Monte Carlo." n.d. https://colindcarroll.com/2019/04/28/choice-of-symplectic-integrator-in-hamiltonian-monte-carlo/.

"Foundations of Linear and Generalized Linear Models." 2015. John Wiley & Sons.

Gelman, Andrew, John B Carlin, Hal S Stern, David B Dunson, Aki Vehtari, and Donald B Rubin. 2013. *Bayesian Data Analysis*. CRC press.

Hoffman, Matthew D, and Andrew Gelman. 2014. "The No-U-Turn Sampler: Adaptively Setting Path Lengths in Hamiltonian Monte Carlo." *J. Mach. Learn. Res.* 15 (1): 1593–1623.

Moore, Jacob. n.d. *MCMC from Scratch*. https://colab.research.google.com/drive/1YQBSfS1Nb8a9TAMsV1RjWsiErWqXLbrj.

Neal, Radford M. 2003. "Slice Sampling." *Annals of Statistics*, 705–41.

Nishio, Motohide, and Aisaku Arakawa. 2019. "Performance of Hamiltonian Monte Carlo and No-U-Turn Sampler for Estimating Genetic Parameters and Breeding Values." *Genetics Selection Evolution* 51 (1): 73.

Thomas, Samuel. 2020. *Hmclearn: Logistic Regression Example*. https://cran.r-project.org/web/packages/hmclearn/.