Hamiltonian Monte Carlo with Energy Conserving Subsampling (HMC-EC)

Markov Chain Monte Carlo - Theory and practical applications

Ancellin Eloi - Eisenbarth Florian - Foucault Armand - Joël Garde

https://arxiv.org/abs/1708.00955

Sommaire

Introduction

- Bayesian Context:
    - Dataset $D \in R^{n*d}$
    - Points coming from law that we do not know how to sample from

- Objective
    - ***HMC-within gibbs*** to sample from a posterior distribution
    - 2 major issues:
        1) Explore the posterior distribution efficiently when ***d*** is large $\Rightarrow$ HMC methods
        2) Keep a reasonable computation cost when **n** is large $\Rightarrow$ Subsampling

1) and 2) at the same time is harder!

Subsampling

- Advantages
  - Driving cost in HMC : evaluation of both the **likelihood** and its **gradient** $\Rightarrow$ $O(n)$
  - Subsample of size **m** $\Rightarrow$ accelerate computation of HMC
  - $O(m) << O(n)$

- Issues:
  - Naive subsampling $\Rightarrow$ Disconnects the Hamiltonien from its own dynamics $\Rightarrow$ Deterioration of energy conserving property $\Rightarrow$ Loses ability to sample efficiently in high dimension

How can we solve problems 1) and 2) at the same time ?

Bayesian inference

- Dataset: $D \in R^{n*d}$, labels $Y \in [-1,1]^n$
- Model $\Phi_\theta(x) = \hat{y}$
- Posterior distribution $\pi(\theta \mid x)$
- Likelihood $L(\theta) = P(x \mid \theta)$
- Prior $\pi(\theta)$
- Bayes Theorem $$\pi(\theta \mid x) = \frac{\pi(\theta)P(x \mid \theta)}{\int \pi(u)P(x \mid u)du}$$

$$\pi(\theta \mid x) \propto \pi(\theta)P(x|\theta)$$

Hamiltonian Monte Carlo (HMC) (1/2)

- Position: $\theta \in \mathbb{R}^d$
- Target distribution: $\pi(\theta) \propto e^{-U(\theta)}$
- Momentum: $p \in \mathbb{R}^d$
- Potential energy: $U(\theta) = \ell(\theta)$ where ℓ is the log-likelihood
- Kinetic energy: $K(p) = \dfrac{p^T M^{-1} p}{2}$
- Extended target density: $\Pi(\theta, p) \propto \exp\{-U(\theta) - K(p)\} \quad p, q \in \mathbb{R}^d$
- The Hamiltonian: $H(\theta, p) = U(\theta) + K(p)$

Hamiltonian Monte Carlo (HMC) (2/2)

- Hamiltonian dynamics:

$$\frac{\partial H}{\partial \theta^{t,i}}(\theta^t, p^t) = \frac{\partial U(\theta^t)}{\partial \theta^{t,i}} = -\frac{dp^{t,i}}{dt}$$
$$\frac{\partial H}{\partial p^{t,i}}(\theta^t, p^t) = \frac{\partial K(p^t)}{\partial p^{t,i}} = p^{t,i} = \frac{d\theta^{t,i}}{dt}$$

- Leapfrog integrator:

$$p^{k+\frac{1}{2}} = p^k - (\frac{h}{2})\nabla U(\theta^k)$$
$$\theta^{k+1} = \theta^k + hM^{-1}p^{k+\frac{1}{2}}$$
$$p^{k+1} = p^{k+\frac{1}{2}} - (\frac{h}{2})\nabla U(\theta^{k+1})$$

Pseudo-Marginal MCMC (1/2)

- Objective: sample from an analytically unknown target distribution $\pi(\theta)$
- Idea: use an unbiased and non-negative estimator :

$$\bar{\pi}(\theta, u) = \bar{\pi}(\theta|u)\, p_U(u)$$

Where *u* is an auxiliary variable with distribution p_U

$\Rightarrow$ $\bar{\pi}$ verifies $E_{p_U}\left[\bar{\pi}(\theta, u)\right] = \pi(\theta)$

Pseudo-Marginal MCMC (2/2)

- In our case, we consider a Bayesian setting, therefore:

$$\bar{\pi}(\theta, u) \propto \hat{L}_m(\theta) p_\Theta(\theta) p_U(u)$$

Where *u* is a subset of our observations and $\hat{L}_m(\theta)$ is an estimator of the likelihood of θ based on the subset *u*. $p_\Theta(\theta)$ is the prior on θ.

⇨ Use the above extended density as the target of a MH algorithm in order to obtain samples θ from π

Hamiltonian Monte Carlo Within Gibbs (1/6)

A “Gibbs framework”, for each iteration, makes alternatively:

- A subsampling step: draw ***u*** among your set of observations *y*
- An HMC step: make a proposal for your new parameter value, according to the Hamiltonian dynamics

Hamiltonian Monte Carlo Within Gibbs (2/6)

$$\bar{\pi}(\theta, p, u) \propto \exp(-\hat{\mathcal{H}}(\theta, p))P_U(u)$$

$$\begin{aligned} u_{k+1} &\sim u \,\big|\, \theta_k, p_k \\ \theta_{k+1}, p_{k+1} &\sim \theta, p \,\big|\, u_{k+1} \end{aligned}$$

Hamiltonian Monte Carlo Within Gibbs (3/6)

1/ Subsample auxiliary variables

- Size *m* of the subsample chosen such that the MCMC walk is computationally efficient:
    - *m* too small → highly unstable estimator (high variance) → low acceptance rate → waste of computational resources
    - *m* too large → too many computations → too much computational time

$$\Rightarrow \quad m = n^{0.5}$$

Hamiltonian Monte Carlo Within Gibbs (4/6)

1/ Subsample auxiliary variables

- Induce correlation within the draws to tolerate higher variance in the estimator
  - By inducing correlation between the *u*'s, we can tolerate a substantially smaller *m*

$\Rightarrow$ Divide *u* into *G* blocks and update only one block at a time randomly $\Rightarrow$ $n = n^{0.5}$

- Propose *u'* according to p_U and accept it with probability

$$\min\left(1, \frac{\hat{L}_m(\theta_{t-1}, u')}{\hat{L}_m(\theta_{t-1}, u_{t-1})}\right)$$

Hamiltonian Monte Carlo Within Gibbs (5/6)

2/ HMC step

- Recall the estimated Hamiltonian:

$$\hat{H}(\theta, p) := \hat{\mathcal{U}}(\theta) + \mathcal{K}(p)$$

- We simulate the Hamiltonian dynamics by leapfrog discretization for *L* steps per HMC step

⇨ We start at $(\theta_0, p_0) = (\theta_{t-1}, p_0)$ and arrive at proposal $(\theta^L, -p^L)$

Hamiltonian Monte Carlo Within Gibbs (6/6)

2/ HMC step

- Accept the proposal with probability

$$\min\left(1, exp\left(-\hat{H}(\theta_L, -p_L) + \hat{H}(\theta_0, p_0)\right)\right)$$

Estimator of the likelihood

	Perturbed	Signed
Unbiased	╳	✔
Positive	✔	╳
Cheap to compute	✔	✔

Difference estimator

$$\hat{h}(\theta) = \sum_{i=0}^{n} q_i(\theta) + \frac{m}{n} \sum_{u_i \in u} \ell_{u_i}(\theta) - q_{ui}(\theta)$$

$\downarrow$

2nd order approximation of the full likelihood

$\downarrow$

Correction based on batch ***u***

Other Tricks

- Dual averaging:
  - Step size automatic.
  - Next step: full blown no-U-turn

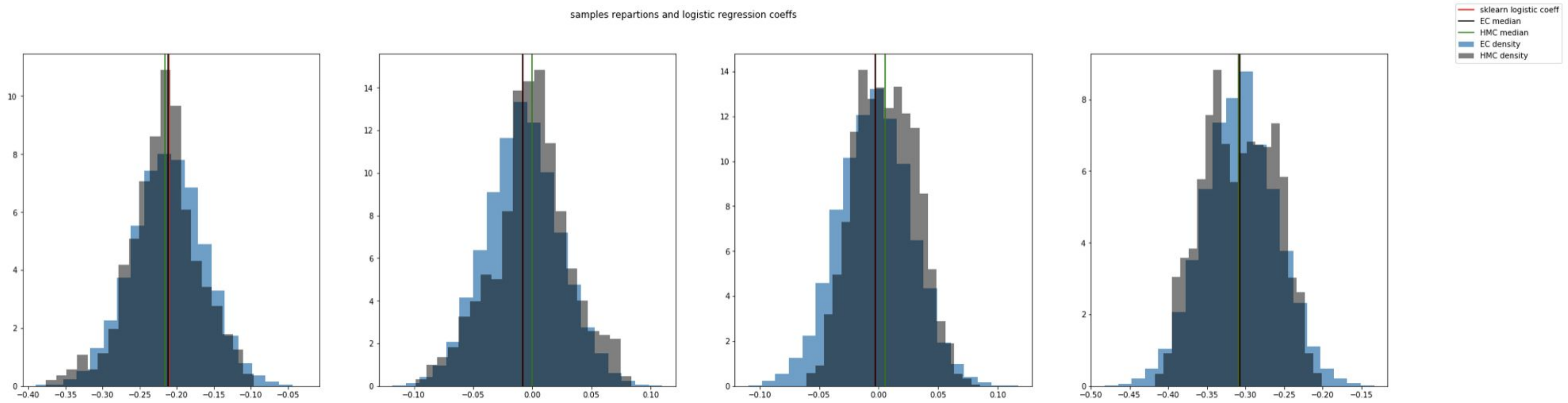
- Approximated optimal Euclidean-Gaussian kinetic energies

$$p \sim \mathcal{N}(0,\ M_{\theta^*}\ )$$

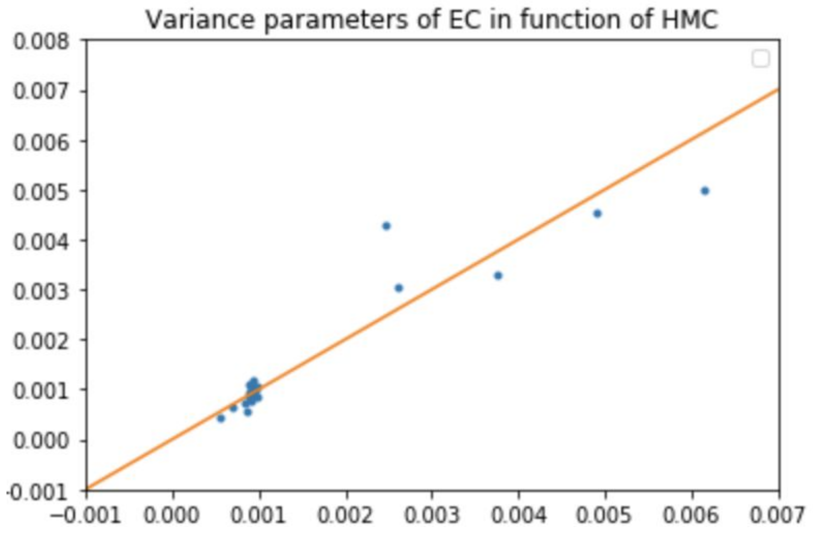
$$M_{\theta\,*} \approx -H_L(\theta^*)\ +\ s^2 I_d$$

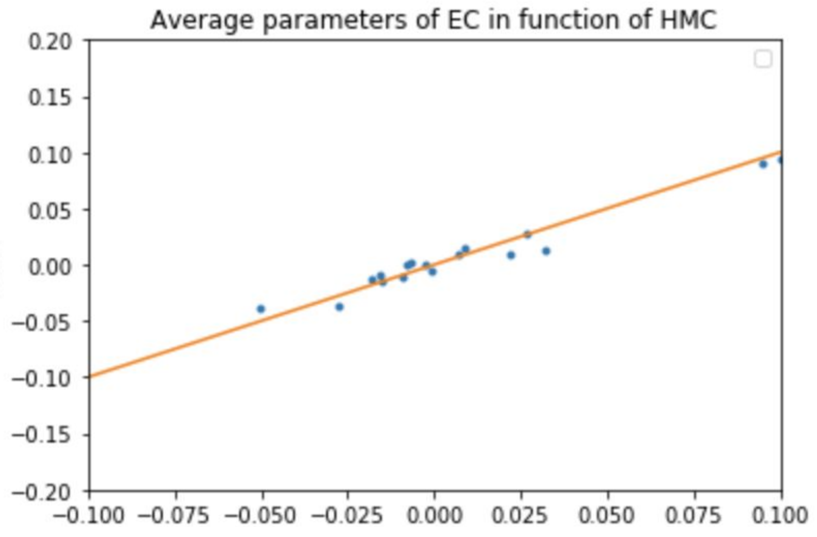
Experimental Results (1/3)

Obtained parameters distribution of HMC(Black) and EC(Blue) for 20 000 iterations.



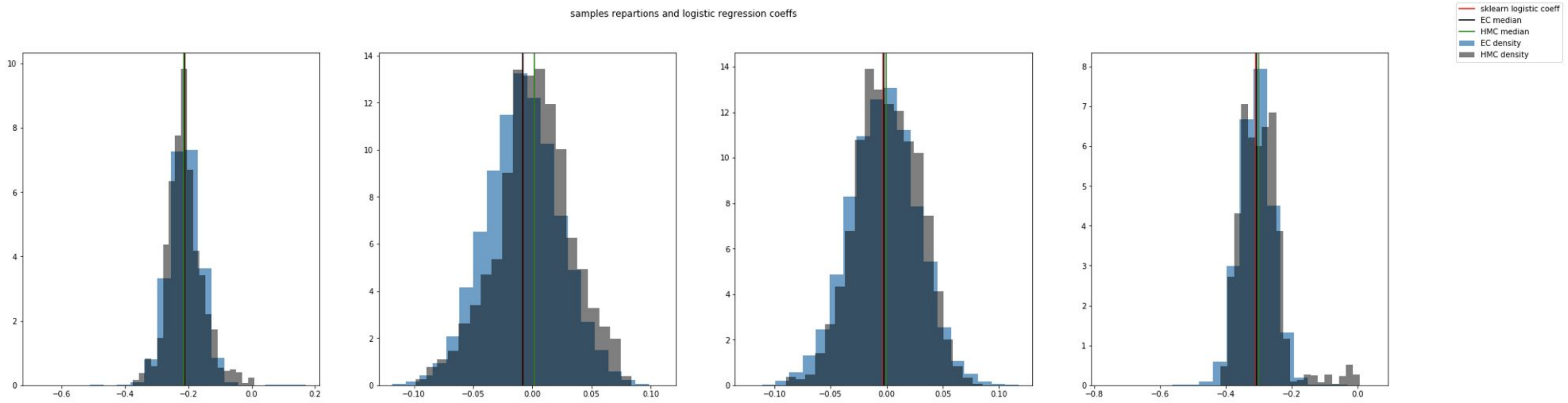
Experimental Results (2/3)





Experimental Results (3/3)

Running the algorithms with 4 000 iterations for the EC and 300 for the HMC (to have a comparable runtime)



Conclusion

Experimental results prove that:

- Inefficiency factor for HMC = 2.08 for EC = 2.185
- Need 5% more iterations for EC than HMC
- EC is 30 times faster than HMC in our measurements (the gain is sqrt(n))

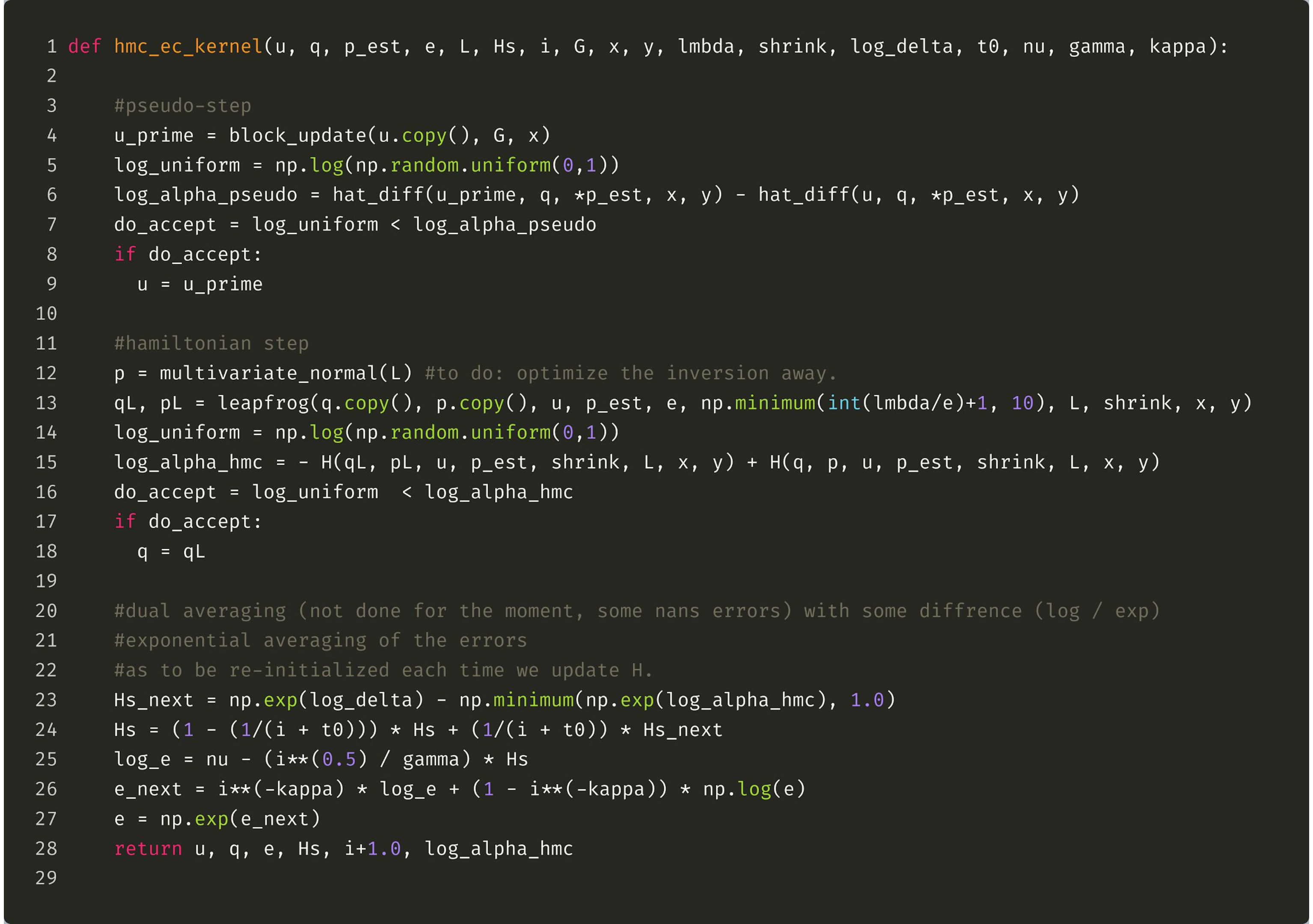
Left to explore:

- No-U-turn to have the good number of leapfrog steps automatically
- Parameter tuning
- Include practical solutions into a theoretical framework (ex: Stepsize lock)
- Parallelize MCMC to fasten the calculations

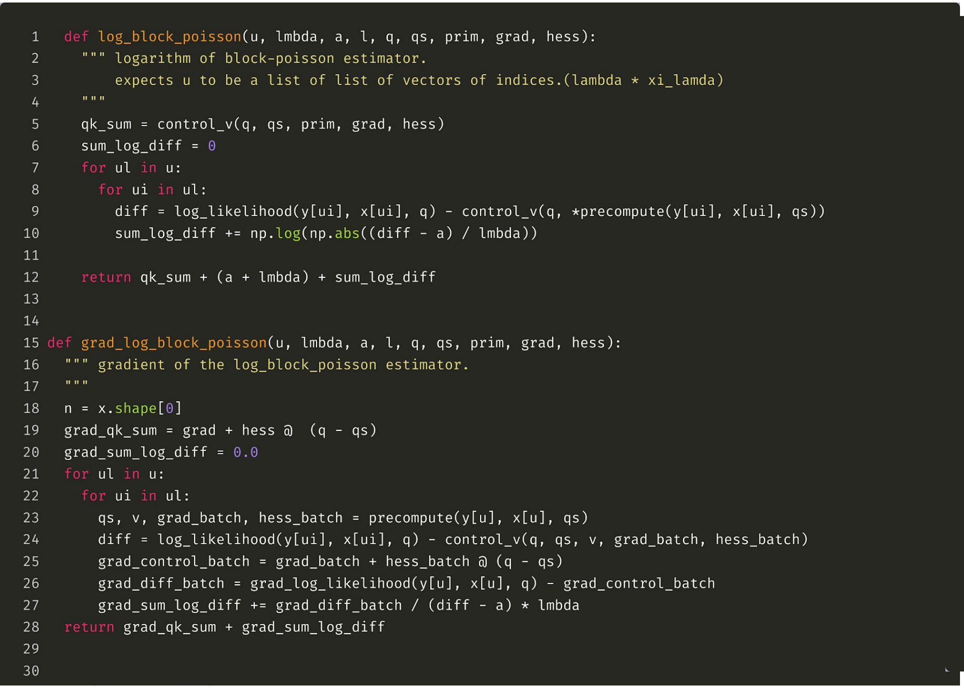
Annexes

Code :

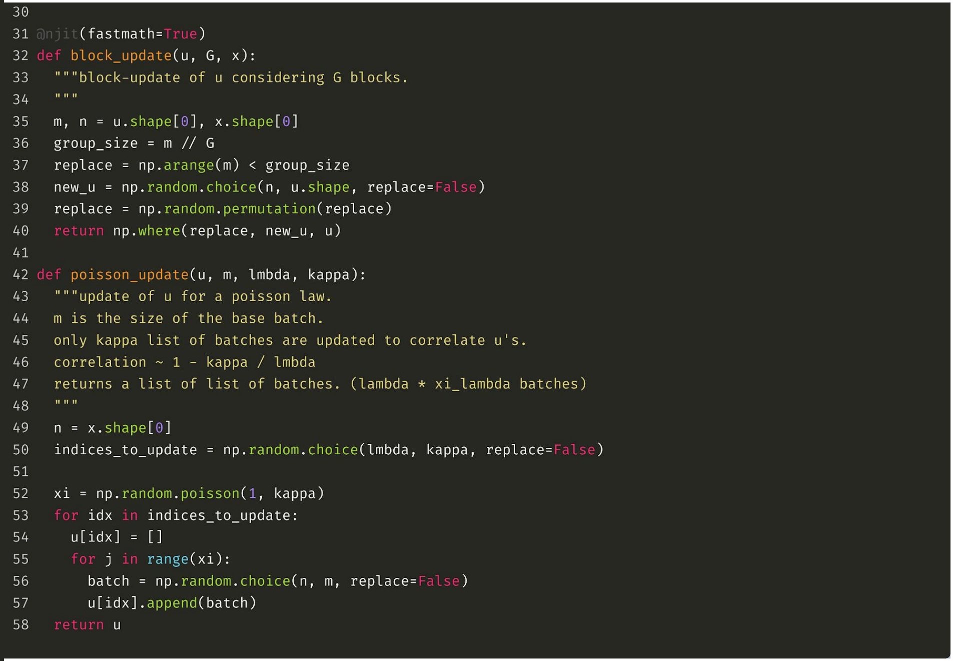
- HMC Energy conservative Kernel
- Leapfrog integrator
- Block poisson
- Log likelihood


```python
def hmc_ec_kernel(u, q, p_est, e, L, Hs, i, G, x, y, lmbda, shrink, log_delta, t0, nu, gamma, kappa):

    #pseudo-step
    u_prime = block_update(u.copy(), G, x)
    log_uniform = np.log(np.random.uniform(0,1))
    log_alpha_pseudo = hat_diff(u_prime, q, *p_est, x, y) - hat_diff(u, q, *p_est, x, y)
    do_accept = log_uniform < log_alpha_pseudo
    if do_accept:
      u = u_prime

    #hamiltonian step
    p = multivariate_normal(L) #to do: optimize the inversion away.
    qL, pL = leapfrog(q.copy(), p.copy(), u, p_est, e, np.minimum(int(lmbda/e)+1, 10), L, shrink, x, y)
    log_uniform = np.log(np.random.uniform(0,1))
    log_alpha_hmc = - H(qL, pL, u, p_est, shrink, L, x, y) + H(q, p, u, p_est, shrink, L, x, y)
    do_accept = log_uniform  < log_alpha_hmc
    if do_accept:
      q = qL

    #dual averaging (not done for the moment, some nans errors) with some diffrence (log / exp)
    #exponential averaging of the errors
    #as to be re-initialized each time we update H.
    Hs_next = np.exp(log_delta) - np.minimum(np.exp(log_alpha_hmc), 1.0)
    Hs = (1 - (1/(i + t0))) * Hs + (1/(i + t0)) * Hs_next
    log_e = nu - (i**(0.5) / gamma) * Hs
    e_next = i**(-kappa) * log_e + (1 - i**(-kappa)) * np.log(e)
    e = np.exp(e_next)
    return u, q, e, Hs, i+1.0, log_alpha_hmc

```

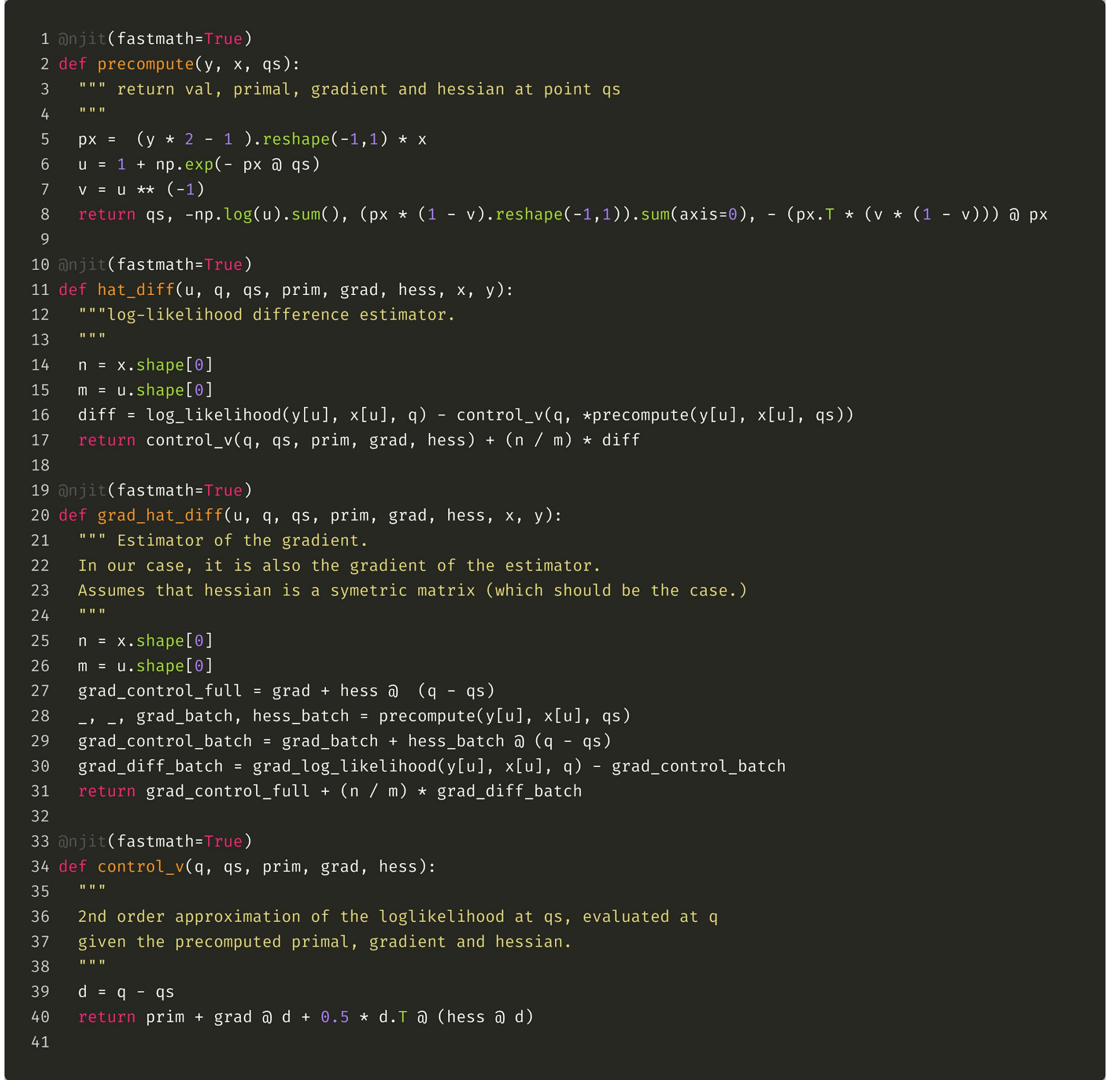
```python
@njit(fastmath=True)
def leapfrog(q, p, u, p_est, e, D, L, shrink, x, y):

  """
  Leapfrog approximator, step size e, length D*e.
  q: cinetic variable.
  p: momemtum variable.
  L: cholesky lower part of M
  grad_U: gradient of the cinetic energy.
  """
  #M_i = np.linalg.inv(L.T @ L) weird, it should be L.T @ L to give me M.
  for i in range(D):
    p -= 0.5 * e * grad_U(q, p, u, p_est, shrink, x, y )
    q += e * np.linalg.solve(L.T, np.linalg.solve(L, p)) #should be doing a fast triangular solve instead. #or just get the inverse since the matrix is small
    p -= 0.5 * e * grad_U(q, p, u, p_est, shrink, x, y)
  return q, -p

```



```python
 def log_block_poisson(u, lmbda, a, l, q, qs, prim, grad, hess):
  """ logarithm of block-poisson estimator.
    expects u to be a list of list of vectors of indices.(lambda * xi_lamda)
  """
  qk_sum = control_v(q, qs, prim, grad, hess)
  sum_log_diff = 0
  for ul in u:
   for ui in ul:
    diff = log_likelihood(y[ui], x[ui], q) - control_v(q, *precompute(y[ui], x[ui], qs))
    sum_log_diff += np.log(np.abs((diff - a) / lmbda))

  return qk_sum + (a + lmbda) + sum_log_diff


def grad_log_block_poisson(u, lmbda, a, l, q, qs, prim, grad, hess):
 """ gradient of the log_block_poisson estimator.
 """
 n = x.shape[0]
 grad_qk_sum = grad + hess @  (q - qs)
 grad_sum_log_diff = 0.0
 for ul in u:
  for ui in ul:
   qs, v, grad_batch, hess_batch = precompute(y[u], x[u], qs)
   diff = log_likelihood(y[ui], x[ui], q) - control_v(q, qs, v, grad_batch, hess_batch)
   grad_control_batch = grad_batch + hess_batch @ (q - qs)
   grad_diff_batch = grad_log_likelihood(y[u], x[u], q) - grad_control_batch
   grad_sum_log_diff += grad_diff_batch / (diff - a) * lmbda
 return grad_qk_sum + grad_sum_log_diff


```



```python

@njit(fastmath=True)
def block_update(u, G, x):
 """block-update of u considering G blocks.
 """
 m, n = u.shape[0], x.shape[0]
 group_size = m // G
 replace = np.arange(m) < group_size
 new_u = np.random.choice(n, u.shape, replace=False)
 replace = np.random.permutation(replace)
 return np.where(replace, new_u, u)

def poisson_update(u, m, lmbda, kappa):
 """update of u for a poisson law.
 m is the size of the base batch.
 only kappa list of batches are updated to correlate u's.
 correlation ~ 1 - kappa / lmbda
 returns a list of list of batches. (lambda * xi_lambda batches)
 """
 n = x.shape[0]
 indices_to_update = np.random.choice(lmbda, kappa, replace=False)

 xi = np.random.poisson(1, kappa)
 for idx in indices_to_update:
  u[idx] = []
  for j in range(xi):
   batch = np.random.choice(n, m, replace=False)
   u[idx].append(batch)
 return u
```



```python
@njit(fastmath=True)
def precompute(y, x, qs):
  """ return val, primal, gradient and hessian at point qs
  """
  px =  (y * 2 - 1 ).reshape(-1,1) * x
  u = 1 + np.exp(- px @ qs)
  v = u ** (-1)
  return qs, -np.log(u).sum(), (px * (1 - v).reshape(-1,1)).sum(axis=0), - (px.T * (v * (1 - v))) @ px

@njit(fastmath=True)
def hat_diff(u, q, qs, prim, grad, hess, x, y):
  """log-likelihood difference estimator.
  """
  n = x.shape[0]
  m = u.shape[0]
  diff = log_likelihood(y[u], x[u], q) - control_v(q, *precompute(y[u], x[u], qs))
  return control_v(q, qs, prim, grad, hess) + (n / m) * diff

@njit(fastmath=True)
def grad_hat_diff(u, q, qs, prim, grad, hess, x, y):
  """ Estimator of the gradient.
  In our case, it is also the gradient of the estimator.
  Assumes that hessian is a symetric matrix (which should be the case.)
  """
  n = x.shape[0]
  m = u.shape[0]
  grad_control_full = grad + hess @  (q - qs)
  _, _, grad_batch, hess_batch = precompute(y[u], x[u], qs)
  grad_control_batch = grad_batch + hess_batch @ (q - qs)
  grad_diff_batch = grad_log_likelihood(y[u], x[u], q) - grad_control_batch
  return grad_control_full + (n / m) * grad_diff_batch

@njit(fastmath=True)
def control_v(q, qs, prim, grad, hess):
  """
  2nd order approximation of the loglikelihood at qs, evaluated at q
  given the precomputed primal, gradient and hessian.
  """
  d = q - qs
  return prim + grad @ d + 0.5 * d.T @ (hess @ d)

```

```python
def log_likelihood(y, x, q):
  """ Likelihood for y = {0, 1}^n
  """
  p = y * 2 - 1  #translate y to -1, 1
  return - np.log(1 + np.exp(- p * (x @ q))).sum()

@njit(fastmath=True)
def grad_log_likelihood(y, x, q):
  p = y * 2 - 1 #translate y to -1, 1
  return (x * ( p * (1 + np.exp(p * (x @ q)))** (-1) ).reshape((-1,1))).sum(axis = 0)

@njit(fastmath=True)
def hessian_log_likelihood(y, x, q):
  p = y * 2 - 1
  return  (p.reshape((-1,1))* x).T @ ( x * (p * ((1 + np.exp( - p * (x @ q)))** (-1))).reshape((-1,1)))
```