

```

1 import autograd.numpy as np
2 from autograd import grad
3 from autograd import elementwise_grad as egrad
4 from autograd.scipy import stats
5 import random
6
7
8 def hamiltonian_monte_carlo(log_density, theta_0, epsilon, leapfrog_iter, overall_iter, n_var=1,
9                             momentum_std=1.0, temperature=1.0, debug=False, log_interval=0):
10     """
11     An implementation of Hamiltonian Monte Carlo, mostly following the pseudocode in the NUTS
12     paper (Hoffman & Gelman, 2011). This function starts from a provided guess of  $\theta$  and
13     generates one chain of samples, without discarding any that might be treated as warmup.
14
15     :param log_density: The log-density to sample from.
16     :param theta_0: An initial guess for the parameters
17     :param epsilon: The step size to take in each leapfrog iteration
18     :param leapfrog_iter: The number of leapfrog iterations to take (often denoted  $L$ )
19     :param overall_iter: The number of overall HMC iterations to take (samples in the chain)
20     :param n_var: The number of variables, the dimension of  $\theta$ , which is assumed to be 1-D
21     :param momentum_std: A parameter for sampling the momentum variable. By default, if a number
22         (or anything that is not callable) is provided, it is treated as  $\sigma$  of a zero-mean
23         normal distribution. If a callable is provided, it is treated as a sampler for momentum.
24     :param temperature: The temperature parameter, which in this case only impacts the acceptance
25         probabilities. Defaults to one.
26     :param debug: Whether or not to include debug prints
27     :param log_interval: How often to log the acceptance counts.
28     :return: A list of samples, of length overall_iter, each a tuple of ( $\theta$ ,  $r$ , accepted)
29     """
30     if log_interval != 0:
31         print(f'HMC: sampling {overall_iter} iterations with {leapfrog_iter} leapfrog iterations with epsilon = {epsilon}')
32
33     samples = []
34     accepted_count = 0
35     theta = np.asarray(theta_0, dtype=np.float64)
36     log_density_grad = grad(log_density)
37     hamiltonian = generate_hamiltonian(log_density)
38
39     # handle the default case, of specifying a zero-mean normal distribution std
40     if not callable(momentum_std):
41         def momentum_sampler():
42             return np.random.normal(0, momentum_std, size=n_var)
43
44     else:
45         momentum_sampler = momentum_std
46
47     if theta.size != n_var:
48         raise ValueError('Theta_0 should have the same number of variables as n_var specifies')
49
50     for i in range(overall_iter):
51         r = momentum_sampler()
52         if n_var == 1:
53             r = r[0]
54
55         new_theta, new_r = leapfrog(theta, r, epsilon, leapfrog_iter, log_density_grad)
56
57         # subtracting inside the exponent for numerical stability
58         acceptance_prob = np.exp((hamiltonian(new_theta, new_r) - hamiltonian(theta, r)) / temperature)
59
60         if debug:
61             print(theta, r, new_theta, new_r, acceptance_prob)
62
63         accepted = np.random.uniform() < acceptance_prob
64
65         if accepted:
66             accepted_count += 1
67             theta = new_theta
68             r = new_r
69
70         samples.append((theta, r, accepted))
71
72         if log_interval > 0 and i % log_interval == 0 and i > 0:
73             print(f'Sampled {i} iterations of which {accepted_count - 1} were accepted')
74
75     return np.asarray(samples)
76
77
78 def leapfrog(theta, r, epsilon, leapfrog_iter, log_density_grad):
79     """
80     A simple implementation of the leapfrog method symplectic integrator.
81     :param theta: The starting point in parameter-space.
82     :param r: The initial momentum.
83     :param epsilon: The step size.
84     :param leapfrog_iter: The number of iterations to take ( $L$ ).
85     :param log_density_grad: The gradient of the log-density, used as the rate
86         of change of the momentum variables (by the partial derivative  $dH/dq$ )
87     :return: The final values for  $\theta$  and  $r$  after taking  $L$  steps
88     """
89     grad_theta = log_density_grad(theta)
90     theta = np.copy(theta)
91     r = np.copy(r)
92     for l in range(leapfrog_iter):

```

```

93     r += epsilon / 2 * grad_theta
94     theta += epsilon * r
95     grad_theta = log_density_grad(theta)
96     r += epsilon / 2 * grad_theta
97
98     return theta, r
99
100
101 def generate_hamiltonian(log_density):
102     """
103     A utility function to generate a Hamiltonian (as a function of $theta$ and $r$)
104     from the log density.
105     :param log_density: The log density to generate a Hamiltonian for.
106     :return: The Hamiltonian function
107     """
108     def h(theta, r):
109         return log_density(theta) - 0.5 * np.dot(r, r)
110
111     return h
112
113
114 def no_u_turn_sampler(log_density, theta_0, epsilon, overall_iter, n_var=1,
115                      momentum_std=1.0, debug=False, log_interval=0):
116     """
117     A straightforward and unoptimized implementation of the No U-Turn Sampler as introduced
118     in Hoffman & Gelman (2011).
119     :param log_density: The log-density to sample from.
120     :param theta_0: An initial guess for the parameters
121     :param epsilon: The step size to take in each leapfrog iteration
122     :param overall_iter: The number of overall HMC iterations to take (samples in the chain)
123     :param n_var: The number of variables, the dimension of $theta$, which is assumed to be 1-D
124     :param momentum_std: A parameter for sampling the momentum variable. By default, if a number
125         (or anything that is not callable) is provided, it is treated as $\sigma$ of a zero-mean
126         normal distribution. If a callable is provided, it is treated as a sampler for momentum.
127     :param debug: Whether or not to include debug prints
128     :param log_interval: How often to log the acceptance counts.
129     :return: A list of samples, of length `overall_iter`, each a tuple of (theta, r, accpeted)
130     """
131     if log_interval != 0:
132         print(f'NUTS: sampling {overall_iter} iterations with epsilon = {epsilon}')
133
134     samples = []
135     theta = np.asarray(theta_0, dtype=np.float64)
136     log_density_grad = grad(log_density)
137     hamiltonian = generate_hamiltonian(log_density)
138
139     # handle the default case, of specifying a zero-mean normal distribution std
140     if not callable(momentum_std):
141         def momentum_sampler():
142             return np.random.normal(0, momentum_std, size=n_var)
143
144     else:
145         momentum_sampler = momentum_std
146
147     if theta.size != n_var:
148         raise ValueError('Theta_0 should have the same number of variables as n_var specifies')
149
150     for i in range(overall_iter):
151         r = momentum_sampler()
152         if n_var == 1:
153             r = r[0]
154
155         u_slice = np.random.uniform(0, np.exp(hamiltonian(theta, r)))
156         theta_left = theta
157         theta_right = theta
158         r_left = r
159         r_right = r
160         j_iter = 0
161         results = set()
162         s_valid = True
163
164         while s_valid:
165             v_direction = (np.random.uniform() < 0.5) and 1 or -1
166             if v_direction == -1:
167                 theta_left, r_left, _, _, new_results, new_s_valid = build_tree(
168                     theta_left, r_left, u_slice, v_direction, j_iter,
169                     epsilon, hamiltonian, log_density_grad)
170
171             else:
172                 _, _, theta_right, r_right, new_results, new_s_valid = build_tree(
173                     theta_right, r_right, u_slice, v_direction, j_iter,
174                     epsilon, hamiltonian, log_density_grad)
175
176             if new_s_valid:
177                 # ugly workaround to the fact that numpy arrays are not valid keys (immutable)
178                 for theta, r in new_results:
179                     results.add((tuple(theta), tuple(r)))
180
181             s_valid = new_s_valid and (np.dot((theta_right - theta_left), r_left) >= 0) and \
182                 (np.dot((theta_right - theta_left), r_right) >= 0)
183
184             if len(results) > 0:
185                 theta, r = random.choice(tuple(results))

```

```

186         theta = np.array(theta)
187         r = np.array(r)
188         accepted = True
189     else:
190         print(f'Encountered an empty result set on iteration {i}')
191         accepted = False
192
193     if debug:
194         print(theta, r)
195
196     samples.append((theta, r, accepted))
197
198     if log_interval > 0 and i % log_interval == 0 and i > 0:
199         print(f'Sampled {i} iterations')
200
201     return np.asarray(samples)
202
203
204 DEFAULT_DELTA_MAX = 1000
205
206
207 def build_tree(theta, r, u_slice, v_direction, j_iter, epsilon, hamiltonian, log_density_grad,
208               delta_max=DEFAULT_DELTA_MAX):
209     """
210     A straightforward and unoptimized implementation of the BuildTree routine of the
211     No U-Turn Sampler as introduced in Hoffman & Gelman (2011).
212     :param theta: The current values for the parameters
213     :param r: The current momentum values
214     :param u_slice: The value of the introduced slice variable
215     :param v_direction: The direction variable
216     :param j_iter: The number of iterations remaining to take
217     :param epsilon: The leapfrog step size to employ
218     :param hamiltonian: The Hamiltonian for the system we're sampling from
219     :param log_density_grad: The gradient of the log density, used for the leapfrog integrator
220     :param delta_max: A maximal error term
221     :return: Edge values of theta and r for both sides of the tree, the set of all valid reached
222             points, and the indicator variable for whether or not we've reached a U-turn (or error)
223     """
224     # base case - take the first step in the direction v
225     if j_iter == 0:
226         new_theta, new_r = leapfrog(theta, r, epsilon * v_direction, 1, log_density_grad)
227
228         results = list()
229         h = hamiltonian(new_theta, new_r)
230         if u_slice < np.exp(h):
231             results.append((new_theta, new_r))
232
233         s_valid = u_slice < np.exp(delta_max + h)
234         return new_theta, new_r, new_theta, new_r, results, s_valid
235
236     # recursive case - build both other side trees.
237     # TODO: consider reframing recursion as for loop?
238     theta_left, r_left, theta_right, r_right, first_results, first_s_valid = build_tree(
239         theta, r, u_slice, v_direction, j_iter - 1, epsilon, hamiltonian, log_density_grad, delta_max)
240
241     # take second step in appropriate direction
242     if v_direction == -1:
243         theta_left, r_left, _, _, second_results, second_s_valid = build_tree(
244             theta_left, r_left, u_slice, v_direction, j_iter - 1, epsilon, hamiltonian, log_density_grad, delta_max)
245     )
246
247     else:
248         _, _, theta_right, r_right, second_results, second_s_valid = build_tree(
249             theta_right, r_right, u_slice, v_direction, j_iter - 1, epsilon, hamiltonian, log_density_grad, delta_max)
250     )
251
252     s_valid = first_s_valid and second_s_valid and \
253         (np.dot((theta_right - theta_left), r_left) >= 0) and \
254         (np.dot((theta_right - theta_left), r_right) >= 0)
255
256     results = first_results + second_results
257     return theta_left, r_left, theta_right, r_right, results, s_valid
258
259
260 def metropolis_hastings(log_density, theta_0, overall_iter, n_var=1,
261                        proposal_std=1.0, temperature=1.0, debug=False, log_interval=0):
262     """
263     A fairly naive implementation of a Metropolis-Hastings sampler, provided mostly for
264     comparison purposes. The support for temperature annealing schedules is the only non-trivial piece.
265     :param log_density: The log-density to sample from.
266     :param theta_0: An initial guess for the parameters
267     :param overall_iter: The number of overall HMC iterations to take (samples in the chain)
268     :param n_var: The number of variables, the dimension of  $\theta$ , which is assumed to be 1-D
269     :param proposal_std: The standard deviation to use when generating Metropolis-Hastings proposals.
270     :param temperature: The temperature parameter, which in this case only impacts the acceptance
271                         probabilities. Defaults to one. If a callable is provided, it is treated as a temperature
272                         function that receives a single parameter, the iteration number. Used to implement a temperature
273                         annealing schedule.
274     :param debug: Whether or not to include debug prints
275     :param log_interval: How often to log the acceptance counts.
276     :return: A list of samples, of length `overall_iter`, each a tuple of (theta, r, accepted)
277     """
278     if log_interval != 0:

```

```
279     print(f'Metropolis-Hastings: sampling {overall_iter} iterations')
280
281     samples = []
282     accepted_count = 0
283     theta = np.asarray(theta_0, dtype=np.float64)
284
285     # handling both a temperature value and a temperature function
286     if not callable(temperature):
287         def temp_func(i):
288             return temperature
289     else:
290         temp_func = temperature
291
292     if theta.size != n_var:
293         raise ValueError('Theta_0 should have the same number of variables as n_var specifies')
294
295     for i in range(overall_iter):
296         new_theta = np.random.normal(theta, proposal_std, size=n_var)
297
298         # subtracting inside the exponent for numerical stability
299         acceptance_prob = np.exp((log_density(new_theta) - log_density(theta)) / temp_func(i))
300
301         if debug:
302             print(theta, new_theta, acceptance_prob)
303
304         accepted = np.random.uniform() < acceptance_prob
305
306         if accepted:
307             accepted_count += 1
308             theta = new_theta
309
310         samples.append((theta, accepted))
311
312     if log_interval > 0 and i % log_interval == 0 and i > 0:
313         print(f'Sampled {i} iterations of which {accepted_count} were accepted')
314
315     return np.asarray(samples)
```