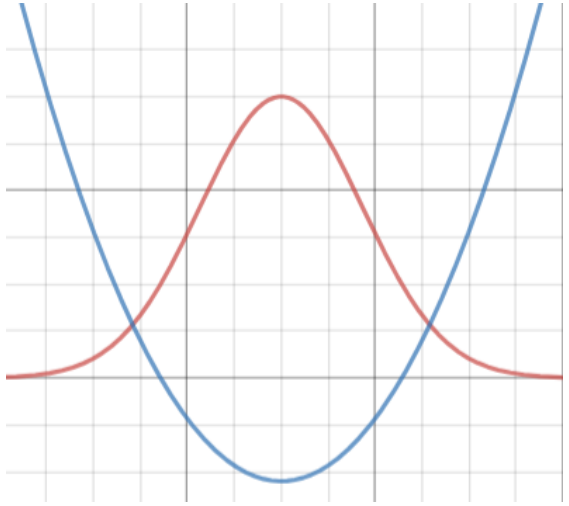
Hamiltonian Monte Carlo

$$p \propto e^{-\frac{E}{kT}} \rightarrow E \propto -\ln p$$

$$q \rightarrow (q, p)$$

(have the momentum component of the chain abide by a stationary distribution from the start)



$$H = U(\boldsymbol{x}) + \frac{1}{2}\boldsymbol{p}^T \boldsymbol{M}\boldsymbol{p}$$

, with $U = -\ln f(x)$ *,* $f(x)$ *the target density*

Evolve along each dimension according to :

$$\frac{dx}{dt} = \frac{\partial H}{\partial p}$$

$$\frac{dp}{dt} = -\frac{\partial H}{\partial x}$$

Accept with probability $a = \min\left(1, \frac{e^{-H(q\prime,p\prime)}}{e^{-H(q,p)}}\right)$

Hamiltonian Monte Carlo Step

1. define the starting position of the proposal $\theta' = \hat{\theta}_t$
2. draw an initial momentum vector p from a multivariate Gaussian distribution: $p \sim \mathcal{N}(0, M)$
3. update the momentum vector by half a step taking the gradient into account: $p' = p - \frac{\epsilon}{2} \cdot \nabla U(\theta')$
4. Repeat for $l = 1, \ldots, L$
   (a) update the position by a full step: $\theta' = \theta' + \epsilon \cdot p'$
   (b) update the momentum by a full step, except at the end of the trajectory: if $(l \neq L)$, then $p' = p' - \epsilon \cdot \nabla U(\theta')$
5. update the momentum vector by half a step: $p' = p' - \frac{\epsilon}{2} \cdot \nabla U(\theta')$
6. negate the momentum vector: $p' = -p'$
7. compute the acceptance probability: $a = \min\left(1, \exp\left[U(\theta_t) - U(\theta') + \frac{\sum p^2}{2} - \frac{\sum p'^2}{2}\right]\right)$
8. set $\theta_{t+1} = \theta'$ with probability a, and $\theta_{t+1} = \theta_t$ otherwise

With:

$$U(\theta_t) = -\log(f(\theta_t))$$

(tune M, ε, L*)*

Daviet, Remi. (2016). Inference with Hamiltonian Sequential Monte Carlo Simulators. SSRN Electronic Journal.

Hamiltonian Monte Carlo Step

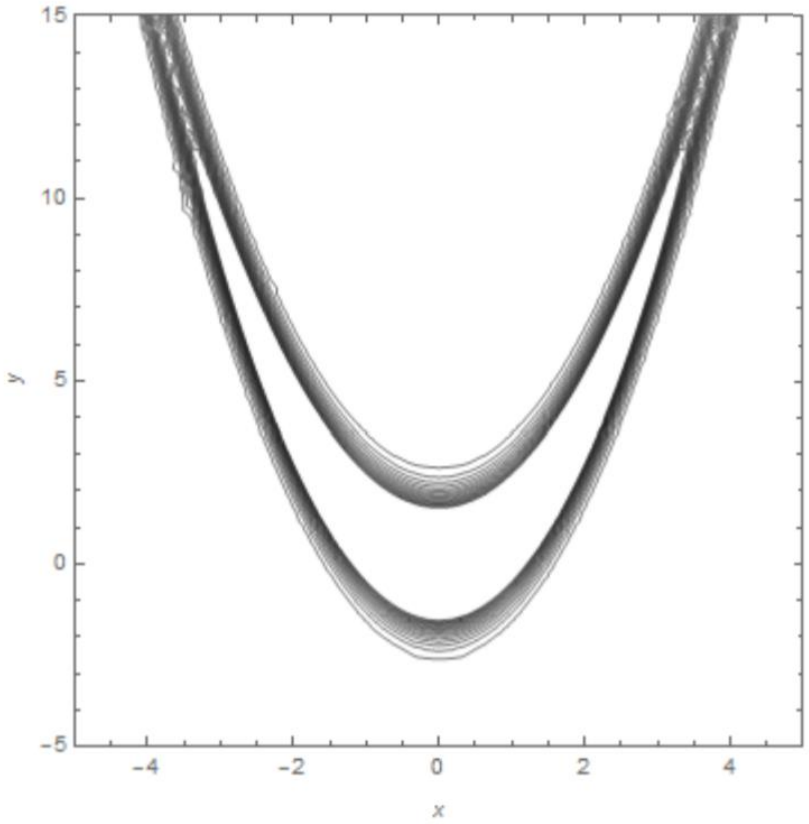
1. define the starting position of the proposal $\theta' = \hat{\theta}_t$
2. draw an initial momentum vector p from a multivariate Gaussian distribution: $p \sim \mathcal{N}(0, M)$
3. update the momentum vector by half a step taking the gradient into account: $p' = p - \frac{\epsilon}{2} \cdot \nabla U(\theta')$
4. Repeat for $l = 1, \ldots, L$
   (a) update the position by a full step: $\theta' = \theta' + \epsilon \cdot p'$
   (b) update the momentum by a full step, except at the end of the trajectory: if $(l \neq L)$, then $p' = p' - \epsilon \cdot \nabla U(\theta')$
5. update the momentum vector by half a step: $p' = p' - \frac{\epsilon}{2} \cdot \nabla U(\theta')$
6. negate the momentum vector: $p' = -p'$
7. compute the acceptance probability: $a = \min\left(1, \exp\left[U(\theta_t) - U(\theta') + \frac{\sum p^2}{2} - \frac{\sum p'^2}{2}\right]\right)$
8. set $\theta_{t+1} = \theta'$ with probability a, and $\theta_{t+1} = \theta_t$ otherwise

- Assume identity mass?
- Negate p step "symbolic"? (acceptance probability at the 7th step already accounts for this inversion, which should guarantee the reversibility of the Markov Chain)

"At the end of the last step, the momentum is reversed to make the proposal symmetric. If we start at the final position with the final reversed momentum, we will find the particle going back to the original position after L steps. This ensures reversibility and facilitate the computation of the acceptance probability."

$$a = \min\left(1, \frac{f(x')g(x|x')}{f(x)g(x'|x)}\right)$$

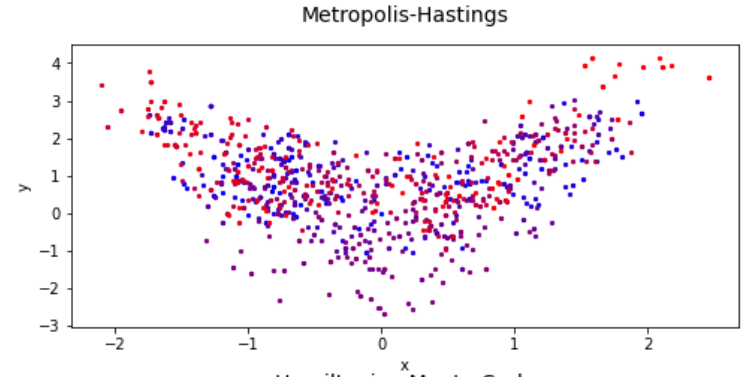
Implementation for a Rosenbrock function

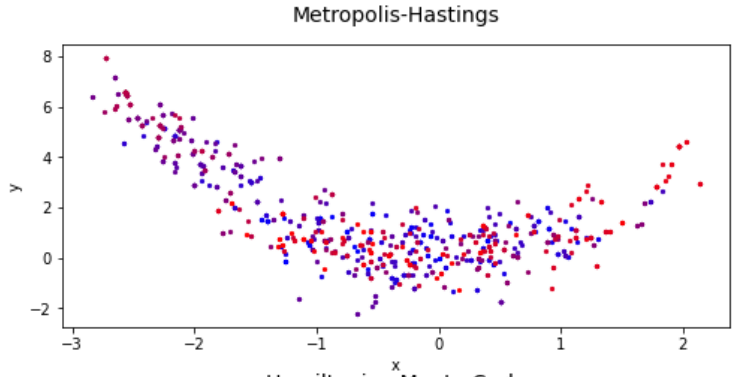


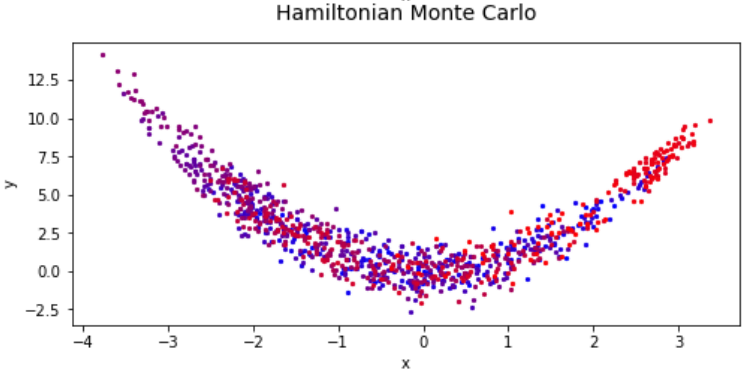
$$f(\theta) \propto e^{\frac{1}{8}\left(-5(y-x^2)^2-x^2\right)}$$

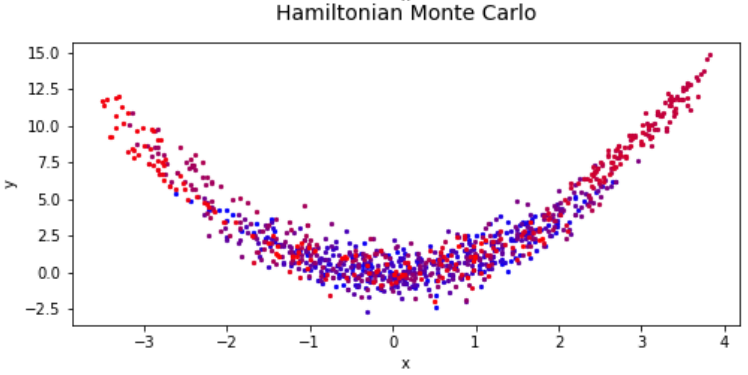
Implementation for a Rosenbrock function

Start the chain at $[0, 0]$









Hamiltonian Monte Carlo:

$M = I_2; \varepsilon = 0.03; \mathrm{L} = 35$

→ Acceptance rate **99%**

(non-HMC) Metropolis-Hastings:

Left:

Transition proposal $\vec{\theta}' \sim \mathcal{N}(\vec{\theta}, 0.2 \cdot I_2)$

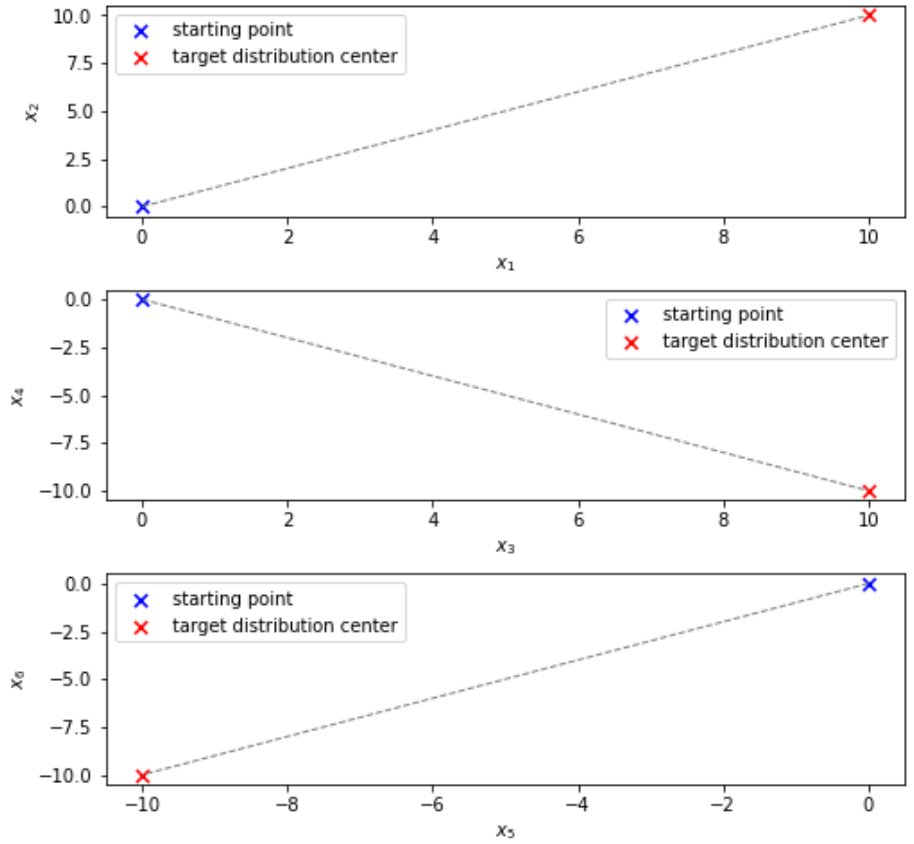
→ Acceptance rate **69%**

Right:

Transition proposal $\vec{\theta}' \sim \mathcal{N}(\vec{\theta}, 1 \cdot I_2)$

→ Acceptance rate **40%**

Implementation for a Multivariate (6-dimensional) Gaussian

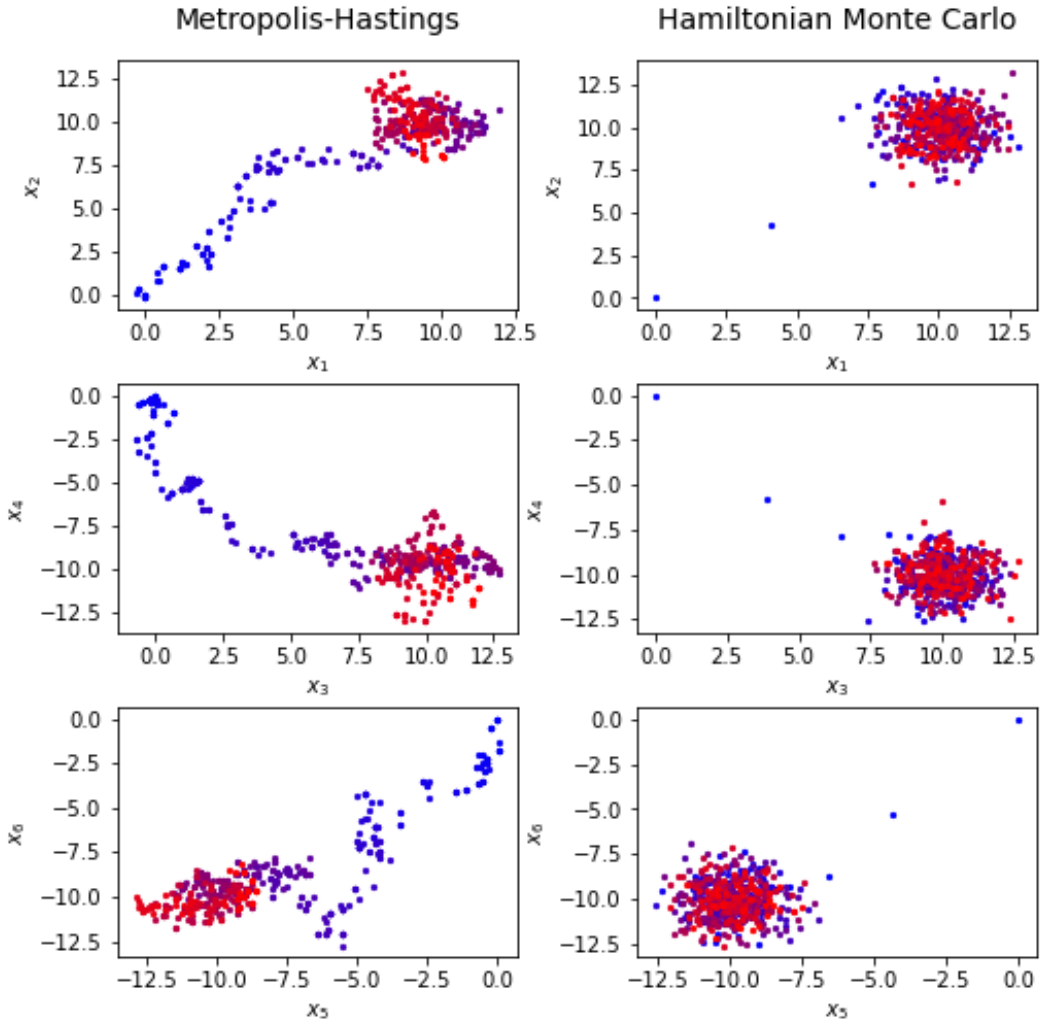


Sample from probability density $\mathrm{f}(\vec{\theta}) \propto \mathcal{N}(\vec{\mu}, I_{6\times 6})$

, with $\vec{\mu} = [10,10,10,-10,-10-10]$

Start the chain at $[0,0,0,0,0,0]$

Paths for 750 steps:



(non-HMC) Metropolis-Hastings:

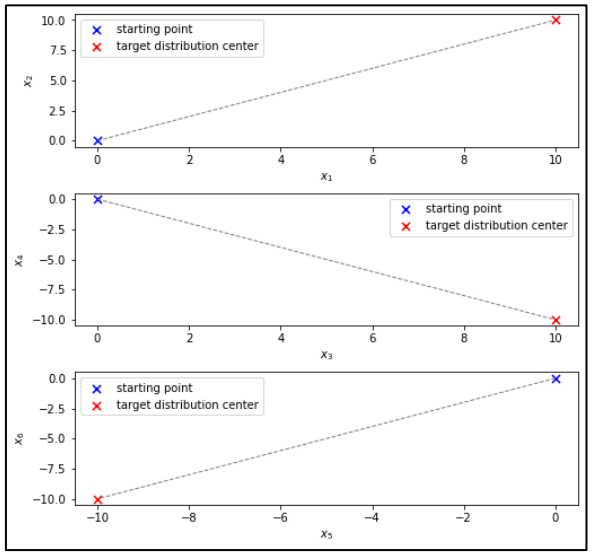
Transition proposal $\vec{\theta}' \sim \mathcal{N}(\vec{\theta}, 0.2 \cdot I_6)$

→ Acceptance rate **56%**

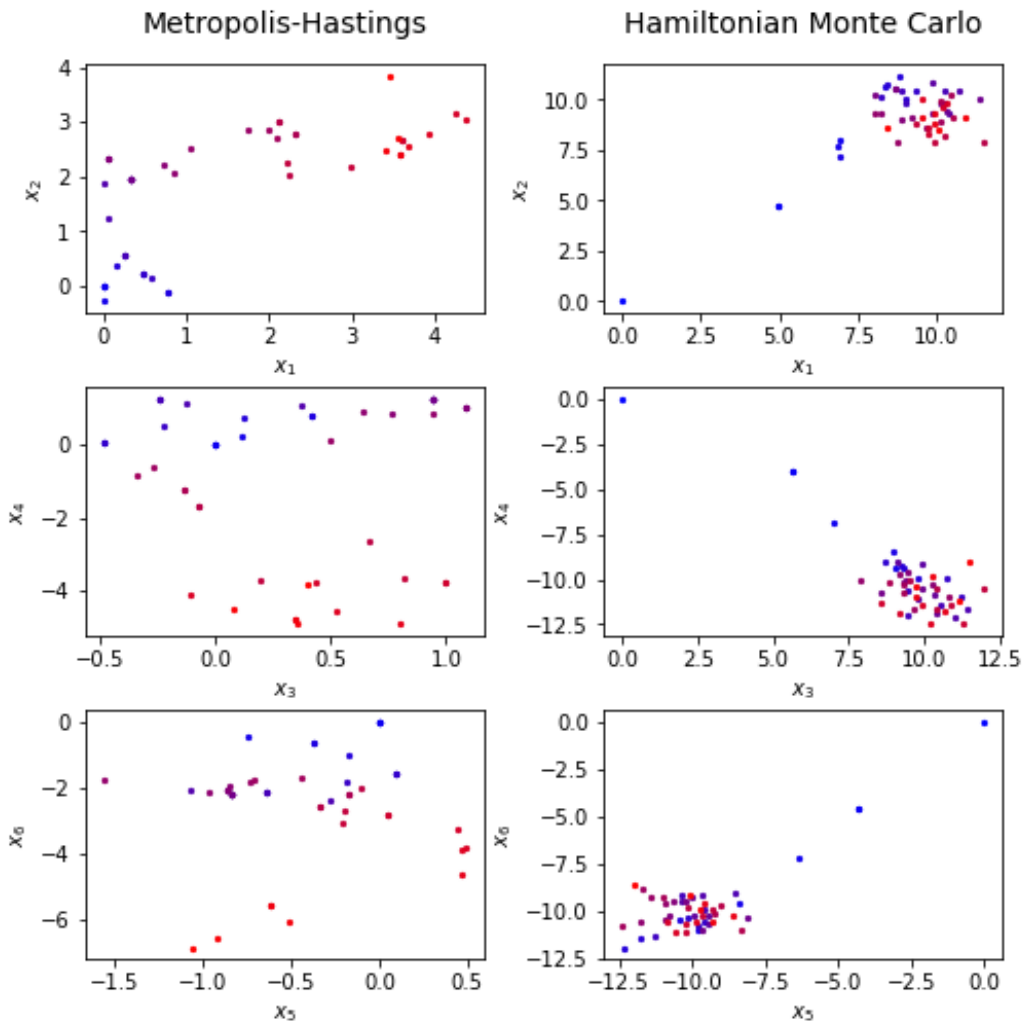
Hamiltonian Monte Carlo:

$M = I_6$; $\varepsilon = 0.05$; $\mathrm{L} = 20$

→ Acceptance rate **99%**



After 50 steps only:



(non-HMC) Metropolis-Hastings:

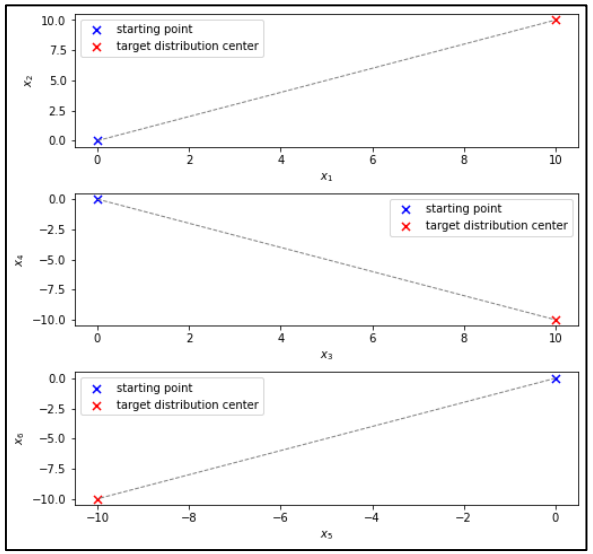
Transition proposal $\vec{\theta}' \sim \mathcal{N}(\vec{\theta}, 0.2 \cdot I_6)$

→ Acceptance rate **60%**

Hamiltonian Monte Carlo:

$M = I_6$; $\varepsilon = 0.05$; $\mathrm{L} = 20$

→ Acceptance rate **98%**



Computing the Gradient $\nabla U(\vec{\theta})$

```python
def simulate_dynamics(initial_momentum, initial_point, L, eta):
    new_point = initial_point
    DU = U_gradient(new_point)
    new_momentum = np.add(initial_momentum,-0.5*eta*DU)
    for l in range(L):
        new_point = np.add(new_point,eta*new_momentum)
        if (l != L-1):
            DU = U_gradient(new_point)
            new_momentum = np.add(new_momentum,-eta*DU)
    new_momentum = np.add(new_momentum,-0.5*eta*DU)
    new_momentum = -new_momentum #?

    p = np.exp(target_U(initial_point)-target_U(new_point)+
            np.sum(initial_momentum**2)/2-np.sum(new_momentum**2)/2)
    return new_point, p
```

In these cases, we know the functions' expressions and can differentiate them.

E.g., for the multivariate Gaussian distribution:

$$\nabla U(\vec{\theta}) = \Sigma^{-1}(\vec{\theta} - \vec{\mu})$$

And when an analytical solution isn't available?

```python
def U_gradient(point,autograd=True):
    if not autograd:
        DU = target_DU(point)
    else:
        DU_f = grad(target_U)
        DU = DU_f(point)
    return(DU)
```

Exact same results, but:

autograd=**False**

→ 2 calls to target + 1 to gradient per step

autograd=**True**

→ 20+ calls to target per step

(offset by the gains from lower correlation - less burn-in /lag samples required? Which auto-differentiation method to choose?)

Sequential Monte Carlo
(with a Metropolis-Hastings mutation step)

1. Initialization: Draw N particles $\{\theta_n^{(0)}\}_{n=1}^N$ from $f_0(\theta_n)$
2. Repeat for $t = 1, \ldots, T$
   (a) Correction: assign weight $w_n^{(t)} = f_t(\theta_n)/f_{t-1}(\theta_n)$ to each of the particles $\{\theta_n^{(t-1)}\}_{n=1}^N$
   (b) Selection: draw N new particles $\{\hat{\theta}_n^{(t)}\}_{n=1}^N$ with replacement from the current sample of particles using weights $w_n^{(t)}$. Give the new particles a weight of 1.
   (c) Mutation: For each particle, perform a MH step as described in section 3.1 to obtain a new sample of particles $\{\theta_n^{(t)}\}_{n=1}^N$.

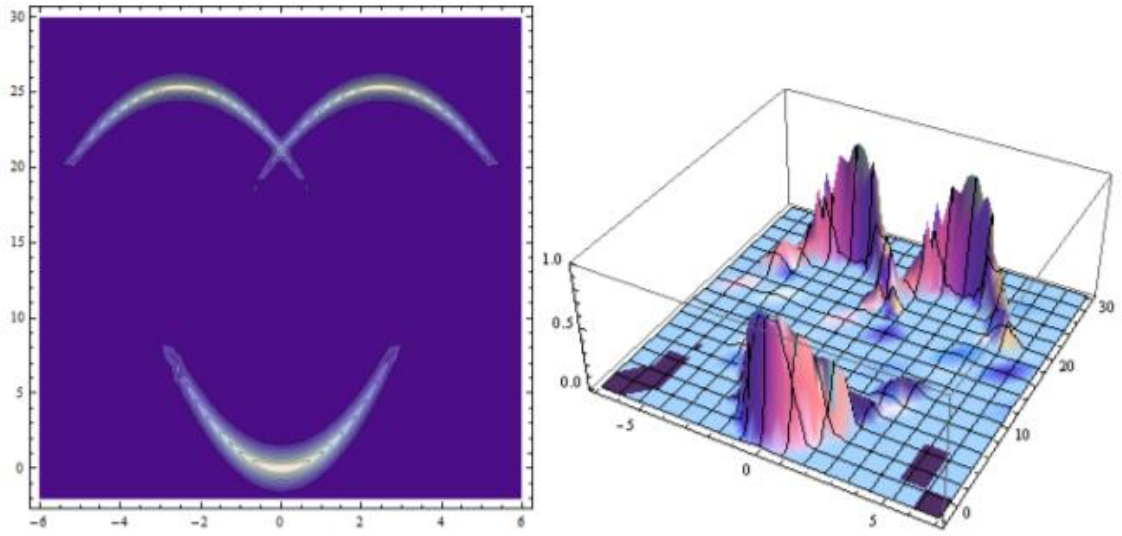
Sequential *Hamiltonian* Monte Carlo

1. Initialization: Draw N particles $\{\theta_n^{(0)}\}_{n=1}^N$ from $f_0(\theta_n)$
2. Repeat for $t = 1, \ldots, T$
   (a) Correction: assign weight $w_n^{(t)} = f_t(\theta_n)/\hat{f}_{t-1}(\theta_n)$ to each of the particles $\{\theta_n^{(t-1)}\}_{n=1}^N$, where $\hat{f}_{t-1}(\theta_n)$ is a "leave-one-out" kernel density estimate.
   (b) Selection: draw N new particles $\{\hat{\theta}_n^{(t)}\}_{n=1}^N$ with replacement from the current sample of particles using weights $w_n^{(t)}$. Give the new particles a weight of 1.
   (c) Mutation: For each particle, perform a Hamiltonian step as described in section 3.2 to obtain a new sample of particles $\{\theta_n^{(t)}\}_{n=1}^N$.

Leave-one-out: so as not to bias the probability estimate for the points which had representation in the discrete distribution (as do all which we re-weight, because they were particles to begin with)?

Smiley Function

Figure 5: contour plot and 3D plot of the smiley function



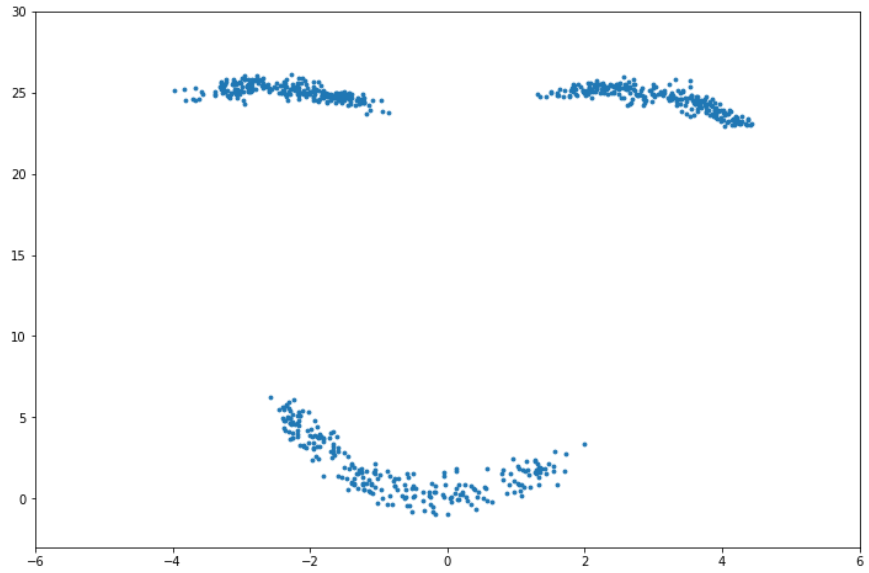
$$f(x,y) \propto e^{\frac{1}{5}\left(-6\left(-(2.5-x)^2-1.5y+38\right)^2-(2.5-x)^2\right)}$$

$$+ e^{\frac{1}{5}\left(-6\left(-(2.5+x)^2-1.5y+38\right)^2-(2.5+x)^2\right)}$$

$$+e^{\frac{1}{5}\left(-5\left(y-x^2\right)^2-x^2\right)}$$

Smiley Function

```
def target(x,i):
    g = []
    g.append(np.exp(1/5*(-6*(-(2.5-x[0])**2-1.5*x[1]+38)**2-(2.5-x[0])**2)))
    g.append(np.exp(1/5*(-6*(-(2.5+x[0])**2-1.5*x[1]+38)**2-(2.5+x[0])**2)))
    g.append(np.exp(1/5*(-5*(x[1]-x[0]**2)**2-x[0]**2)))
    return (g[i])
```

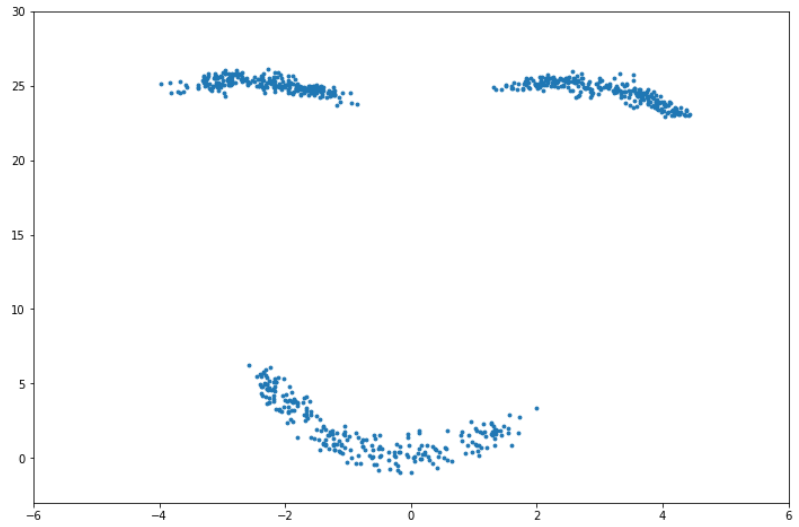


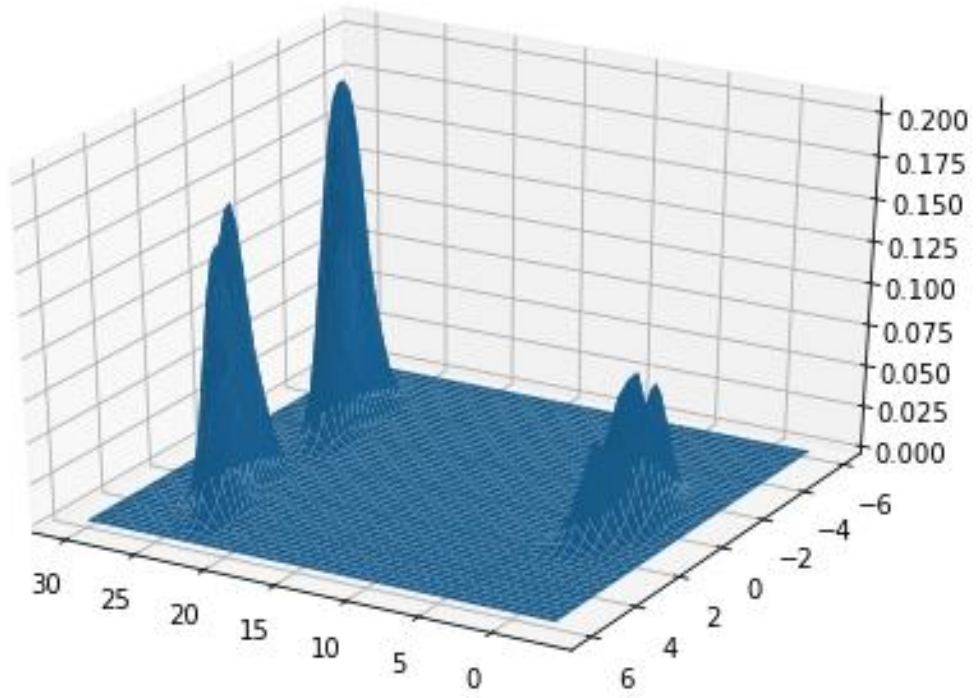

```
def hamiltonian_MC_path(points,i,burn_in=10,lag=5,start=initial_point):
    if (i>=2):
        lag=20
    path = []
    path.append(np.array(start))
    for b in range(burn_in):
        path[0]=hamiltonian_MC_step(path[0],i)
    for t in range(1,points):
        path.append(hamiltonian_MC_step(path[t-1],i))
        for l in range(lag):
            path[t]=hamiltonian_MC_step(path[t],i)
    print("HMC: %d%% particle acceptance rate. " % (100*accepted/total))
    return path
```

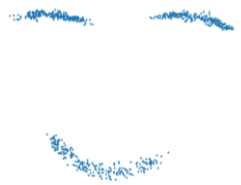
```
starts=[]
starts.append(np.array([2.,25.]))
starts.append(np.array([-2,25.]))
starts.append(np.array([0.,1.]))

points=[]
for i in range(3):
    points.append(hamiltonian_MC_path(256,i,start=starts[i]))
points=points[0]+points[1]+points[2]
```

Smiley Kernel Density Estimate

```
def kernel_density_estimate(x, points, leave_out=None):
    if leave_out is None:
        n = len(points)
    else:
        n = len(points)-1
    h = n**-0.2
    kde = 0
    for point in points:
        if not np.array_equal(point,leave_out):
            kde += multivariate_gaussian(x,point,h*np.identity(dim),
                                         normalize=True)/(n*h)
    return(kde)
```



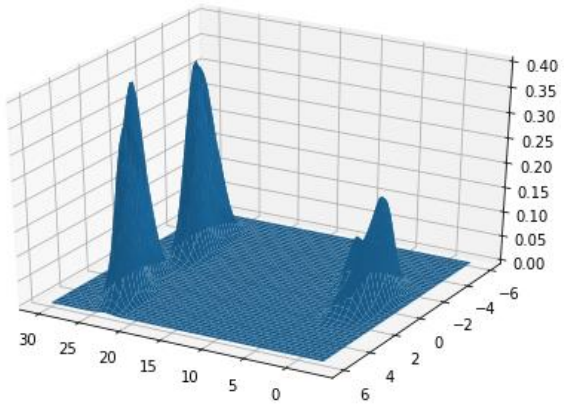




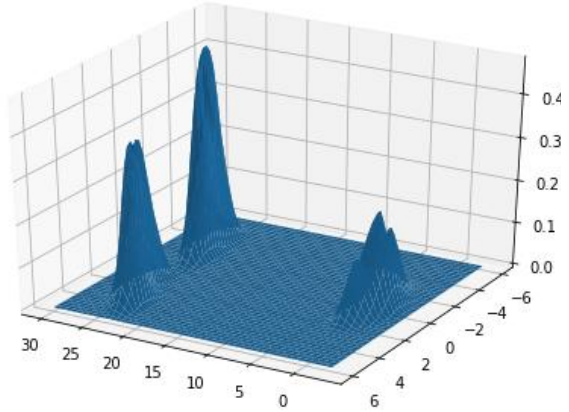
Smiley Kernel Density Estimate

```
points=[]
for i in range(3):
    points.append(hamiltonian_MC_path(256,i,start=starts[i]))
points=points[0]+points[1]+points[2]

points = shuffle_split_accumulate(points)

with open('smileyface3.data', 'wb') as filehandle:
    pickle.dump(points, filehandle)
```

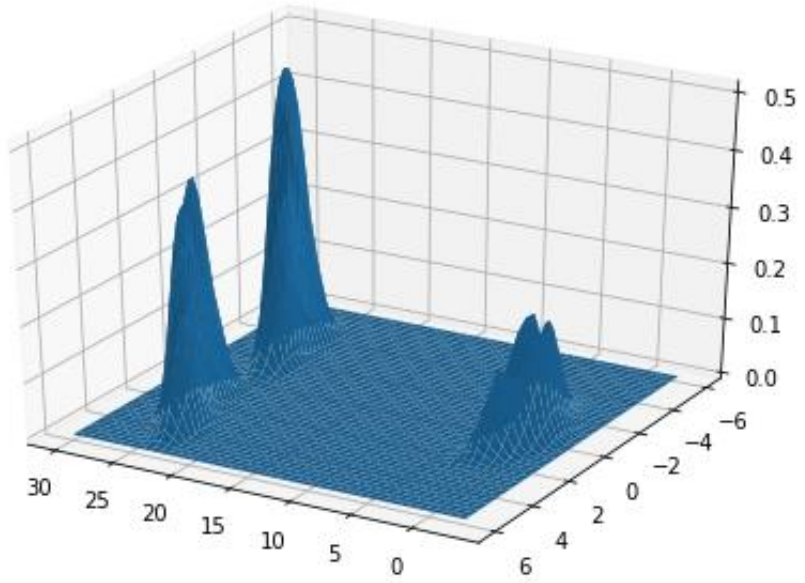
```
def shuffle_split_accumulate(arr,chunksize=100):
    np.random.shuffle(arr)
    arr = [arr[i:i+chunksize] for i in range(0, len(arr), chunksize)]
    arr = list(accumulate(arr))
    return(arr)
```



Datachunk **1** of 8 (100 points)



Datachunks 1 to **4** of 8 (400 points)



Datachunks 1 to **8** of 8 (768 points)
(all data)

Smiley Kernel Density Estimate

SMC

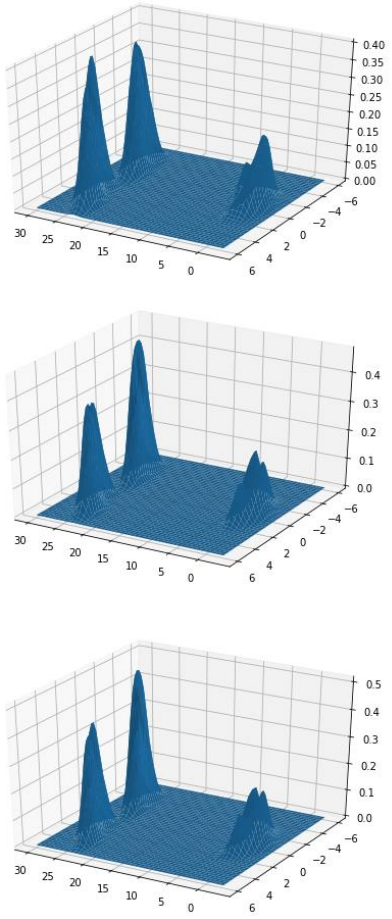
```
def sequentialMC_MH(n_particles,data):
    means = np.array([0,10])
    Sigma = np.matrix([[10,0],[0,20]])
    particle_list = np.random.multivariate_normal(means,Sigma,n_particles)
    particles = {}
    for particle in particle_list:
        key = particle.tobytes()
        particles[key] = 1

    for t in range(len(data)):
        # Correction step.
        for key in particles:
            particle = np.frombuffer(key,dtype='float64')
            if (t==0):
                particles[key] = target(particle,data[t])\
                    /multivariate_gaussian(particle,means,Sigma)
            else:
                particles[key] = \
                    target(particle,data[t])/target(particle,data[t-1])
        # Selection and mutation steps.
        selection_and_mutation(particles,data[t])
    print("\nSequential Monte Carlo (MH)): %d%% particle acceptance rate. " %
          (100*accepted/total))
    key_list = list(particles.keys())
    particles = [np.frombuffer(key,dtype='float64') for key in key_list]
    return particles
```

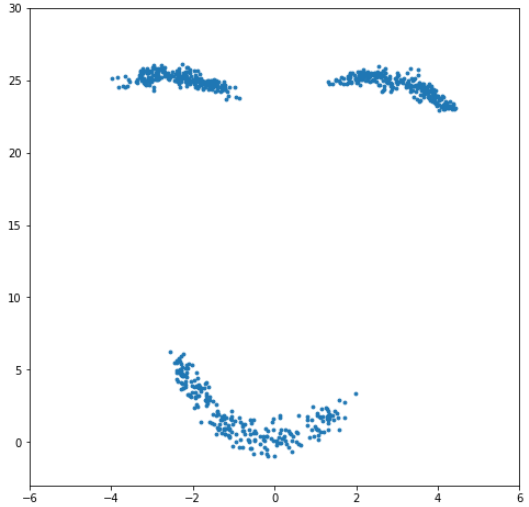
```
def selection_and_mutation(particles,datachunk):
    n = len(particles)
    selected_particles = random.choices(list(particles.keys()),
                                        weights=particles.values(), k=n)
    particles.clear()
    for key in selected_particles:
        particle = np.frombuffer(key,dtype='float64')
        repeated = True
        while (repeated == True):
            mutated_particle = metropolis_hastings_step(particle,datachunk)
            if (mutated_particle.tobytes() not in particles):
                repeated = False

        particles[mutated_particle.tobytes()] = 1
```

SHMC

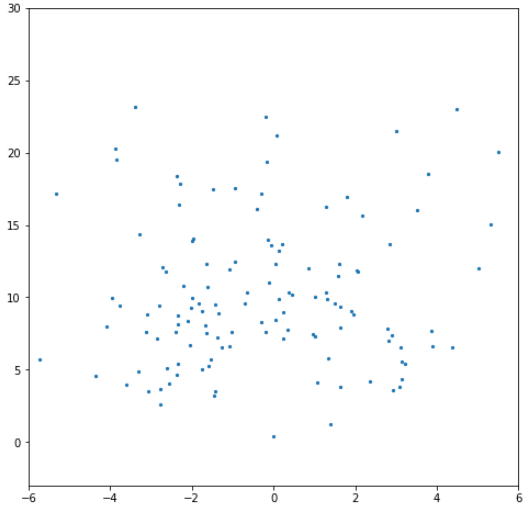
```
def sequentialMC_MH(n_particles,data):
    means = np.array([0,10])
    Sigma = np.matrix([[10,0],[0,20]])
    particle_list = np.random.multivariate_normal(means,Sigma,n_particles)
    particles = {}
    for particle in particle_list:
        key = particle.tobytes()
        particles[key] = 1

    for t in range(len(data)):
        # Correction step.
        for key in particles:
            particle = np.frombuffer(key,dtype='float64')
            if (t==0):
                particles[key] = target(particle,data[t])\
                    /multivariate_gaussian(particle,means,Sigma)
            else:
                particles[key] = \
                    target(particle,data[t])/target(particle,data[t-1])
        # Selection and mutation steps.
        selection_and_mutation(particles,data[t])
    print("\nSequential Monte Carlo (MH)): %d%% particle acceptance rate. " %
          (100*accepted/total))
    key_list = list(particles.keys())
    particles = [np.frombuffer(key,dtype='float64') for key in key_list]
    return particles
```

```
def selection_and_mutation(particles):
    n = len(particles)
    selected_particles = random.choices(list(particles.keys()),
                                        weights=particles.values(), k=n)
    particles.clear()
    for key in selected_particles:
        particle = np.frombuffer(key,dtype='float64')
        repeated = True
        while (repeated == True):
            mutated_particle = hamiltonian_MC_step(particle)
            if (mutated_particle.tobytes() not in particles):
                repeated = False
        particles[mutated_particle.tobytes()] = 1
```

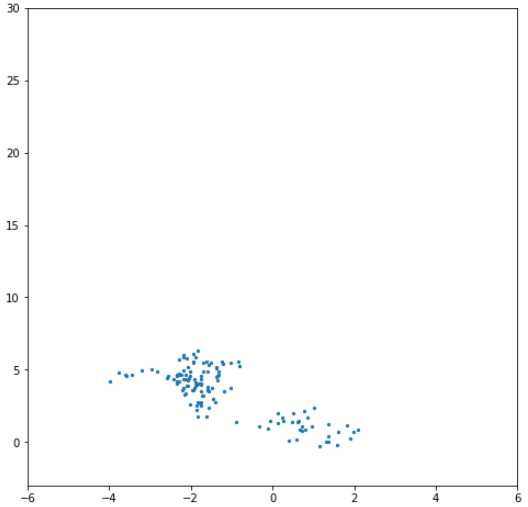




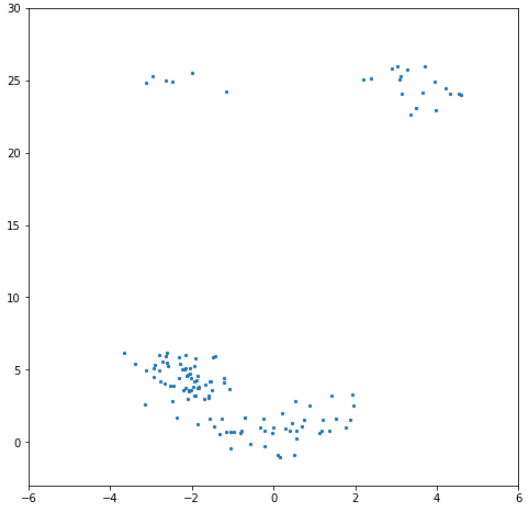
Original
3x256 particles
Shuffled and split into 8* datachunks for the cumulative KDEs
*7 with 100 particles, 8th with 68



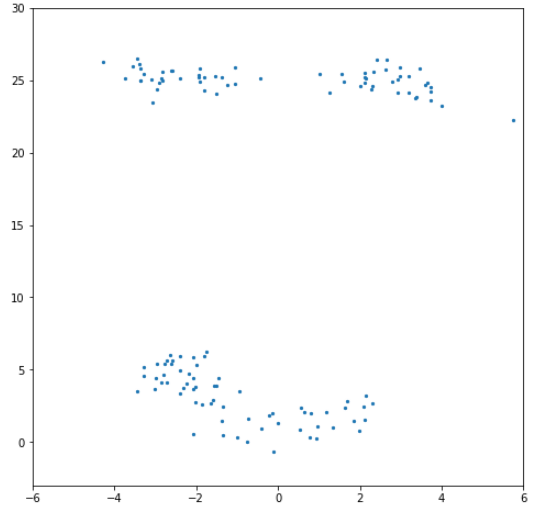
Parallel MH
128 particles
steps (on final density)
$\sigma = 0.05$
66% acceptance



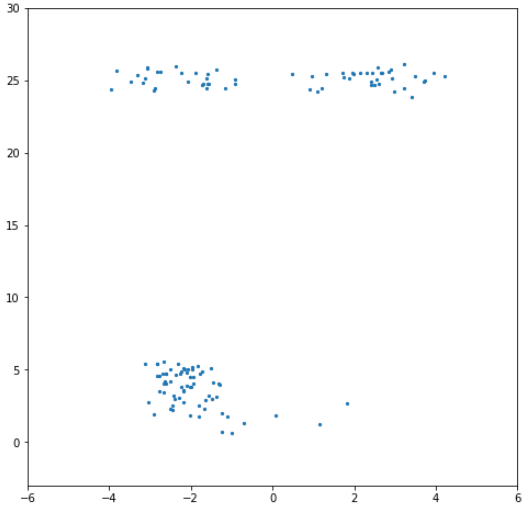
Sequential MH
4x32 particles
8 steps
$\sigma = 0.05$
78% acceptance



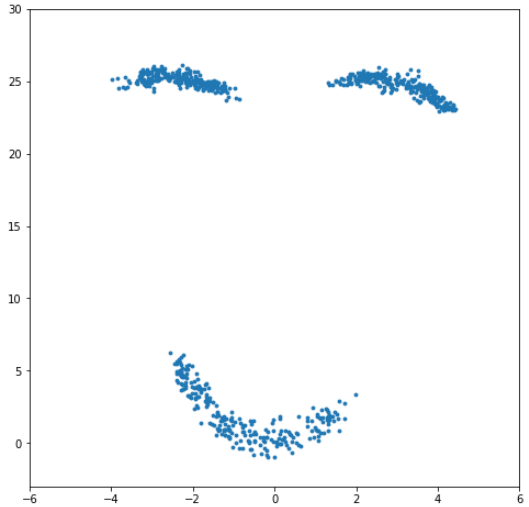
Parallel HMC
128 particles
3 steps (on final density)
$(M, L, \varepsilon) = (I, 20, 0.05)$
96% acceptance



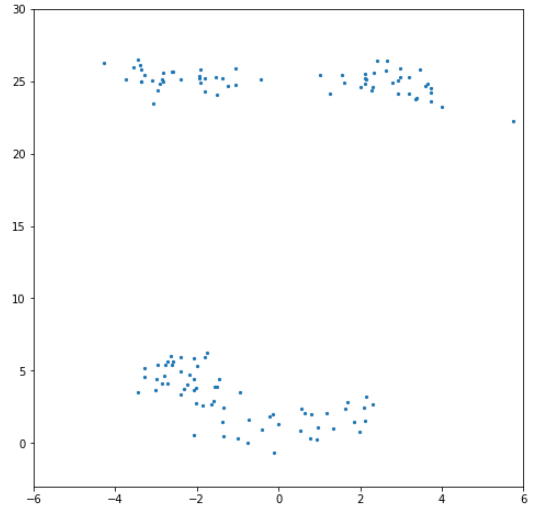
Sequential HMC
4x32 particles
8 steps
$(M, L, \varepsilon) = (I, 20, 0.05)$
99% acceptance



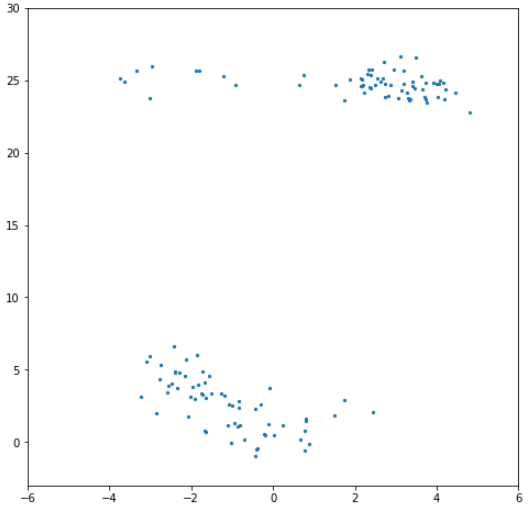
Sequential HMC
with leave-one-out KDE
4x32 particles
8 steps
$(M, L, \varepsilon) = (I, 20, 0.05)$
96% acceptance



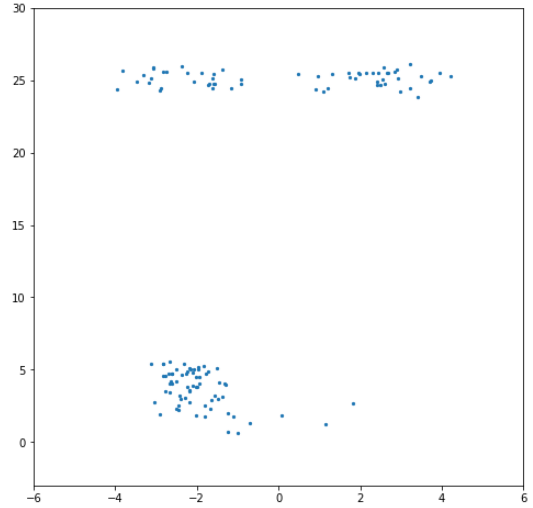
Original
(data for the KDE)
3x256 particles



Sequential HMC
using only the theoretical function
4x32 particles
8 steps
$(M, L, \varepsilon) = (I, 20, 0.05)$
99% acceptance



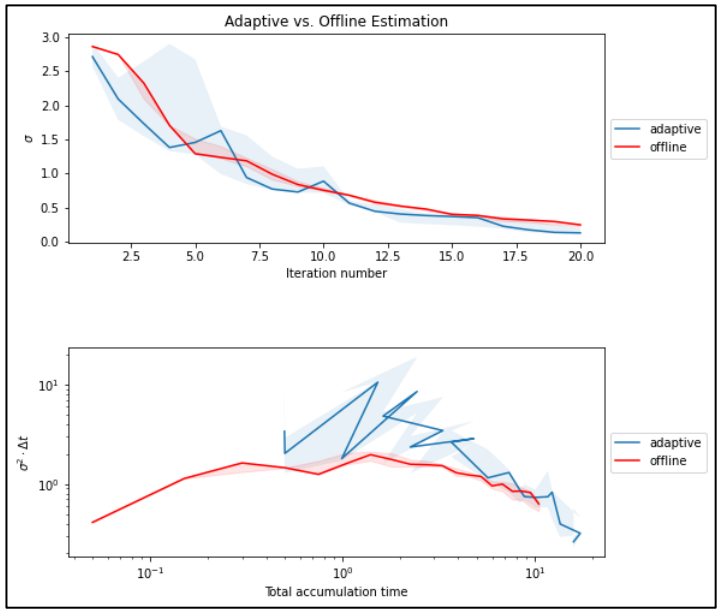
Sequential HMC
with **complete** KDE
4x32 particles
8 steps
$(M, L, \varepsilon) = (I, 20, 0.05)$
96% acceptance

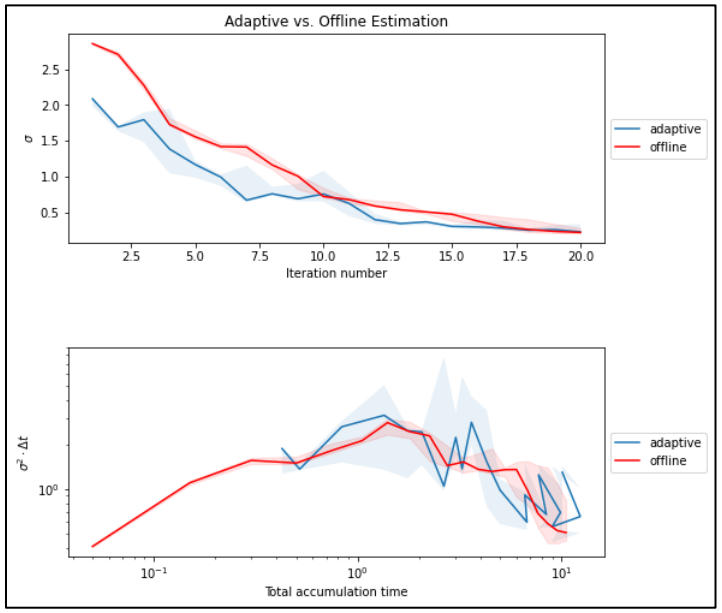


Sequential HMC
with leave-one-out KDE
4x32 particles
8 steps
$(M, L, \varepsilon) = (I, 20, 0.05)$
96% acceptance

SMC vs. CSHMC in the Precession Example

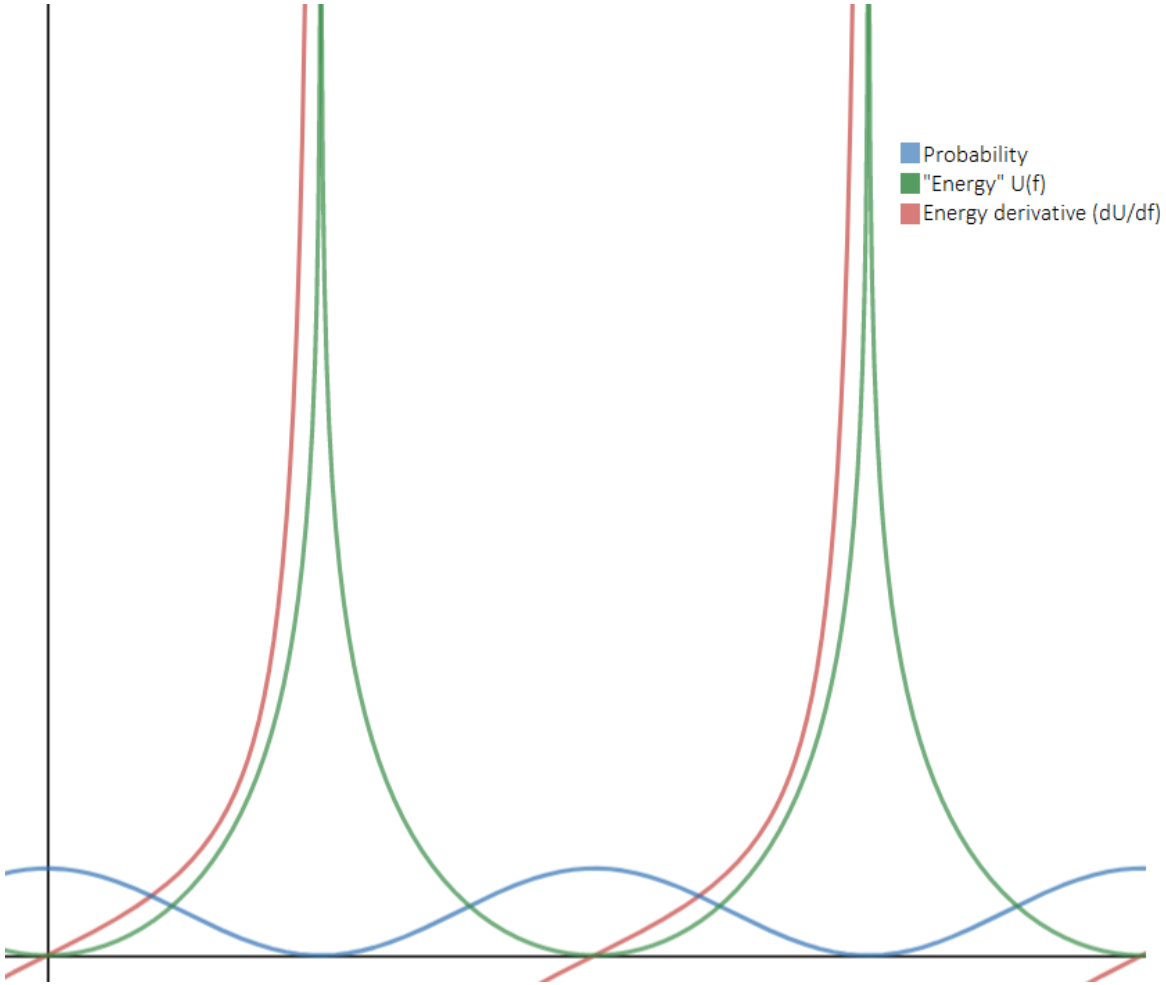
(1d, 1000 particles, median over 5 runs)





$\mathrm{M} = 1, \mathrm{L} = 100, \varepsilon = 10^{-7}$

Doesn't work well for some values of t and ω, requiring too low ε and too high L for reasonable convergence speeds



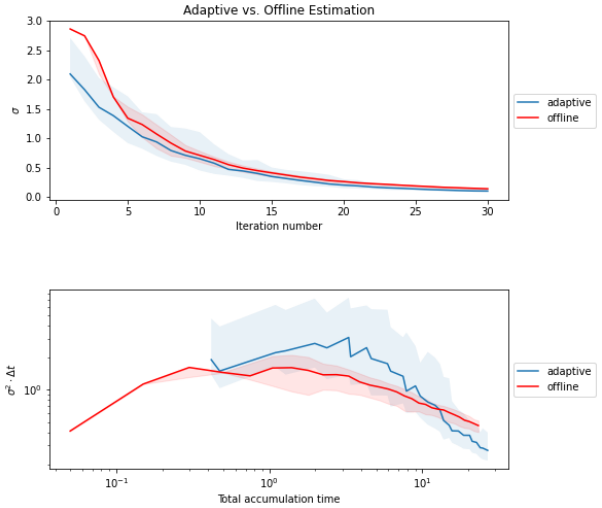
Leapfrog integration error becomes too large when $t\omega \rightarrow k\pi$?

Leads to low acceptance probability, and particles get "trapped"; rejecting repeated particles may then cause the computation not to finish, which is likely when particles with high importance weights lie close to the assymptotes, making the probability of acceptance so low for the usual parameters that the HMC step is practically deterministic (i.e. the particle never moves).

Choosing a very small ε will not completely correct this, plus it will unnecessarily spend resources when points are far from these regions (which is almost always)

Using Metropolis-Hastings steps when the acceptance probability for HMC crosses some threshold:

SMC

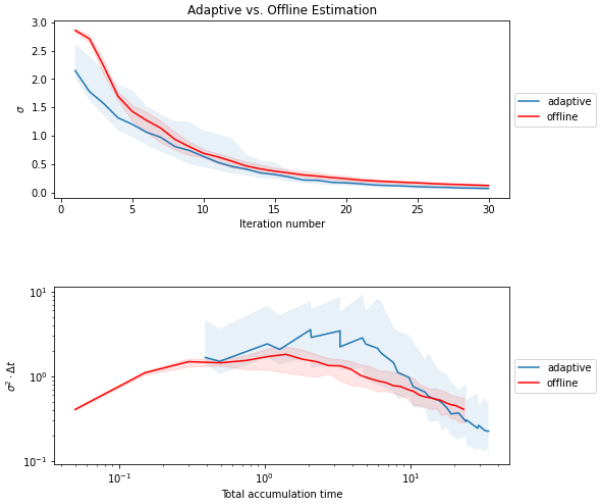


Adaptive:
- Standard deviation: **0.10**
- Error: 0.10
- Final precision: **0.27**

Offline:
- Standard deviation: **0.14**
- Error: 0.11
- Final precision: **0.46**

All: n=1000; N=30; f_max=10; alpha=0.00; median over 100 runs with randomly picked true values
Adaptive: k=0.7, single guess per step
Offline: increment=0.08

SHMC



Adaptive:
- Standard deviation: **0.08**
- Error: 0.07
- Final precision: **0.23**

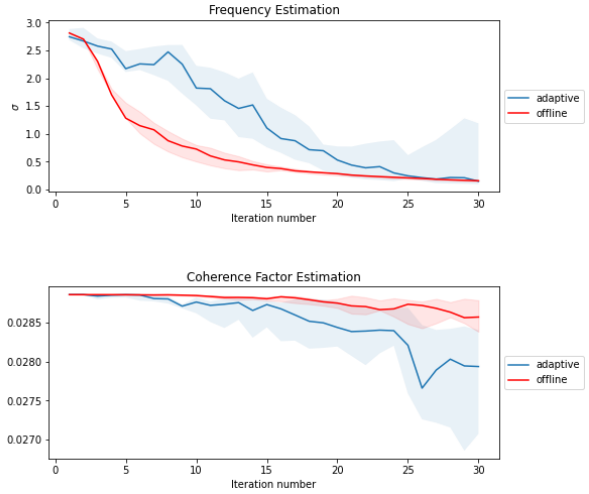
Offline:
- Standard deviation: **0.13**
- Error: 0.11
- Final precision: **0.41**

HMC: $\mathrm{M} = \mathrm{variance}, \mathrm{L} = 20, \varepsilon = 10^{-4}$
MH : $\sigma = \mathrm{variance}$
* Percentage of HMC steps: >**99.9**%.
* Hamiltonian Monte Carlo steps: **99**% mean particle acceptance rate.
* Metropolis-Hastings steps: **89**% mean particle acceptance rate.

(The probability of acceptance for MH is high due to its being used where the derivative is steep, which means starting points come from regions of close to zero probability; this close to zero probability will be in the denominator for the acceptance probability)

And for the 2-dimensional case:

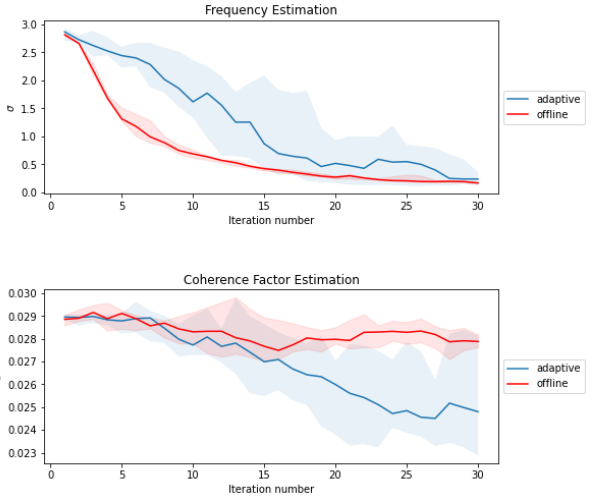
SMC



Adaptive:
- Standard deviation: **0.20**; **0.028**
- Error: 0.16; 0.03
- Final precision: 257.03

Offline:
- Standard deviation: **0.16**; **0.028**
- Error: 0.06; 0.06
- Final precision: 186.45

SHMC



Adaptive:
- Standard deviation: **0.13**; **0.027**
- Error: 0.10; 0.02
- Final precision: 258

Offline:
- Standard deviation: **0.19**; **0.028**
- Error: 0.05; 0.05
- Final precision: 184

All: n=2500; N=30; f_max=10; alfa_max=0.1; median over 10 runs with randomly picked true values
Adaptive: k=3.5, single guess per step
Offline: increment=0.08

HMC: $\mathrm{M} = \mathrm{Cov}, \mathrm{L} = 50, \varepsilon = 10^{-6}$
* Percentage of HMC steps: **100%**.
* Hamiltonian Monte Carlo steps: **100%** mean particle acceptance rate.