Individual Seminar Report for RSCAM

Stein Variation Gradient Descent

Report on Stein Variation Gradient Descent as developed and presented in a paper by Q.Liu & D.Wang, 2019: Stein variational gradient descent: A general purpose bayesian inference algorithm

James, Zoryk

January 26, 2024

This report introduces and explores the Stein Variational Gradient Descent (SVGD) algorithm as a powerful and general-purpose tool for Bayesian Inference. Unlike conventional methods such as Markov Chain Monte Carlo (MCMC) [1, 2], SVGD leverages variational inference [3] through deterministic optimization, providing efficiency and scalability for handling large datasets. The algorithm is rooted in the theoretical connection between the Kernelized Stein Discrepancy (KSD) [4] and the Kullback-Leibler (KL) divergence, making it applicable beyond variational inference. This report outlines the algorithm, delves into its theoretical foundations, and offers insights into its practical implementation.

Background

Kernels and Feature spaces

In SVGD, kernels and feature spaces play a key role in facilitating efficient computation with high-dimensional data. Let $X \subseteq \mathbb{R}^d$ denote the input space. A function $k : X \times X \to \mathbb{R}$ is considered a kernel function if there exists an inner product space $\langle F, \langle \cdot, \cdot \rangle \rangle$ and a function $\varphi : X \mapsto F$ such that

$$k(x, x') = \langle \varphi(x), \varphi(x') \rangle$$

holds for all $x, x' \in X$. This concept is employed to find linear solutions for problems in the feature space F that may not be solvable through linearity in X. In many instances, the function φ is unknown and unnecessary, although theorems exist asserting its existence [5, 6]. In simpler terms, the kernel function k can be interpreted as a measure of similarity between two points. Consider the Gaussian Radial Basis Function (RBF) kernel as an example, defined by

$$k(x, x') = \exp\left(1/h \|x - x'\|_2^2\right) \tag{1}$$

for a suitable choice of $h \in \mathbb{R}$, known as the bandwidth or scale parameter. The transition to the higher-dimensional function space becomes evident by examining the classic example depicted in the code 1 with the results plotted in Figure 1. On the left side of the figure, the input data is challenging to categorize. However, when the Gaussian RBF (1) is applied to the data, categorization becomes trivially feasible through the use of a linear hyperplane.

Kullback-Leibler divergence

The Kullback-Leibler (KL) divergence serves as a measure of how different one continuous probability distribution is from another. Given two separate probability distributions $p(x)$ and $q(x)$ over the random variable x, the KL divergence [7] is given by

$$KL\left(p||q\right) = \mathbb{E}_{x\sim p}\left[\log \frac{p(x)}{q(x)}\right] = \mathbb{E}_{x\sim p}\left[\log p(x) - \log q(x)\right].$$

A noteworthy property of the KL divergence is its non-negativity, being equal to zero if and only if $p(x)$ and $q(x)$ are the same distribution. In SVGD, the KL divergence serves as an 'error measure' between two probability densities, quantifying the discrepancy between the current approximation and the target distribution.

Kernel Stein Discrepancy KSD

The foundational theoretical framework for SVGD is comes from the concept of Kernelized Stein Discrepancy (KSD) [4]. KSD combines the potency of Stein's Identity [8] with the principles of Reproducing Kernel Hilbert Space (RKHS) theory [5]. This results in a computationally tractable measure of discrepancy between a ground truth distribution and samples from an arbitrary distribution.

Stein Identity

Consider a smooth density function $p(x)$ defined on the space X, and let $\phi(x) = [\phi_1(x), \phi_2(x), \ldots, \phi_d(x)]^T$ be a smooth vector function. Stein's Identity [8] states that for a smooth density $p(x)$, the expected value of the Stein operator $\mathcal{A}_p$ acting on $\phi(x)$ is zero:

$$\mathbb{E}_p \left[\mathcal{A}_p \phi(x)\right] = 0 \quad \text{where} \quad \mathcal{A}\phi(x) := \phi(x)\nabla_x \log p(x)^T + \nabla_x \phi(x).$$

Here we call $\mathcal{A}$ as the *Stein Operator*, which is a function operator. Notably, $\mathcal{A}_p$ can often be computed in practice, even for models with intractable normalization constants, owing to its dependency only on p through the score function $\nabla_x \log p(x)$. The linearity of the Stein operator, expressed as $\mathcal{A}(f+g) = \mathcal{A}f + \mathcal{A}g$, is a key property.

Stein Discrepancy

Now, if we consider the expedition is taken with respect to q, while $\mathcal{A}_p$ is still related to p, then there must exist a function ϕ such that it holds

$$\begin{aligned} \mathbb{E}_{x\sim q}\left[\mathcal{A}_p\phi(x)\right] &= \mathbb{E}_{x\sim q}\left[\mathcal{A}_p\phi(x)\right] - \mathbb{E}_{x\sim q}\left[\mathcal{A}_q\phi(x)\right] &(2)\\ &= \mathbb{E}_{x\sim q}\left[\left(\nabla_x \log p(x) - \nabla_x \log q(x)\right)\phi(x)^T\right]. &(3) \end{aligned}$$

Equation (3) represents the differences between the score functions, implying non-zero values if and only if $p = q$. Moreover, it can be demonstrated [9] that this leads to

$$\mathbb{E}_{x\sim q}\left[\mathcal{A}_p\phi(x)\right] = \mathbb{E}_{x\sim q}\left[\text{trace}(\mathcal{A}_p\phi(x))\right]$$

where the trace is employed to form a scalar, though other suitable matrix norms can be used. This leads to the definition of the Stein Discrepancy measure:

$$\mathbb{S}(q,p) = \max_{\phi\in\mathcal{H}^d, ||\phi||\le 1} \left\{\mathbb{E}_p\left[\text{trace}(\mathcal{A}_p\phi(x))\right]^2\right\}. \tag{4}$$

Here, we apply the condition that $\mathcal{H}^d$ is a RKHS for some kernel k. It has been shown [4] that the optimal solution for Equation (4) is given by

$$\phi^*_{q,p}(\cdot) = \mathbb{E}_{x\sim q}\left[\mathcal{A}_p k(x,\cdot)\right]. \tag{5}$$

This result forms the cornerstone of the SVGD algorithm, empowering it with the ability to efficiently estimate the discrepancy between probability distributions.

Stein Variation Gradient Descent

It has been has been shown [10] that under a deterministic transition $z = T(x) + \varepsilon\phi(x)$ with $x \sim q(x)$ and using the ideas from Equation (2) can lead to

$$\nabla_\varepsilon \, \text{KL}(q_{[T]}||p)\Big|_{\varepsilon=0} = -\mathbb{E}_{x\sim q}\left[\text{trace}(\mathcal{A}_p\phi(x)\right]. \tag{6}$$

By utilising the change of variable formula for probability distribution, the distribution of $q_{[T]}$ can be expressed as

$$q_{[T]}(z) = q\left(T^{-1}(z)\right) \cdot \det\left|\nabla_z T^{-1}(z)\right|.$$

Using the results and concepts discussed in the previous sections, it can be shown that Equation (5) represents the optimal direction of perturbation in the unit ball of the RKHS $\mathcal{H}^d$. This direction minimizes the KL divergence of the transformed distribution $q_{[T]}$ optimally, with the magnitude change given by:

$$\nabla_\varepsilon \mathrm{KL}(q_{[T]}||p) = -\mathbb{S}(q, p) \tag{7}$$

In practice, applying this identity perturbation at every time step ε reduces the KL divergence by a factor of $\varepsilon\mathbb{S}(q, p)$. Iterating this process for a sufficient number of steps leads to convergence to the true distribution p. This process can be expressed as an Ordinary Differential Equation:

$$\partial_t x = \hat{\phi}^*_{q,p}(x)$$

where $\hat{\phi}^*_{q,p}(x)$, representing the expectation as stated in Equation (5), can be empirically estimated by taking the mean of n sampled particles:

$$\hat{\phi}^*_{q,p}(x) = \frac{1}{n}\sum_{j=1}^{n}\left[k(x_j, x)\nabla_{x_j}\log p(x_j) + \nabla_{x_j}k(x_j, x)\right].$$

Hence we can construct an iterative update for each particle x_i by the scheme define in [10]

$$x_i \leftarrow x_i + \frac{\varepsilon}{n}\sum_{j=1}^{n}\left[k(x_j, x_i)\nabla_{x_j}\log p(x_j) + \nabla_{x_j}k(x_j, x_i)\right] \tag{8}$$

which is implemented in the Algorithm 1, which again is extracted from [10]. Furthermore, the term $k(x_j, x)\nabla_{x_j}\log p(x_j)$ can be interpreted as the force that drives the particles towards the regions of high probability in of the $p(x)$ distribution. On the other hand, the term $\nabla_{x_j}k(x_j, x)$ acts as a repulsive force between x_j and x_i which push the particles away from each other, ensuring the particles do not bunch together and sample the distribution more evenly. If we consider the case for a single particle $n = 1$ then Equation (8) reduces to the standard gradient descent for maximizing $\log p(x)$, which is referred to as the MAP. This is due to $\nabla_x k(x, x) = 0$, which holds for RBF (1). With the addition of more particles, SVGD interpolates between gradient descent and approximate inference. Two different methods have been presented to analyze SVGD theoretically. The first treats SVGD as the gradient flow of the KL divergence function in the space of probability measures metrized by a RKHS variation of the Wasserstein distance [11]. Alternatively, in [12], fixed points of SVGD are viewed as a type of quadratic quadrature method, from which an approximation bound can be derived.

Numerical Experiments

A simplified version of Algorithm 1 was implemented (see Code 2). The kernel was set to the Radial Basis Function (RBF) as in (1) with a bandwidth $h = 1$ for simplicity. The step size $\varepsilon = 0.01$ was chosen for a balance between accuracy and convergence speed. In [10], AdaGrad was used, and in [13], a coin betting algorithm determined an optimal step size. A toy example (see code 3) was set up with the target distribution $p(x) = \mathcal{N}(x \; -4, 1/2) + \mathcal{N}(x; 2, 4)$, a bimodal Gaussian distribution. The iteration began with 100 particle samples from $\mathcal{N}(0, 1)$, visible in the histogram plot in Figure 2. Subsequent iterations (Figures 3 and 4) showcase convergence to the bimodal distribution of $p(x)$. Another example (see code 4) set the target as a two-dimensional Gaussian $p(x) = \mathcal{N}\left((2,0)^T,\, ((2,0),(0,1))^T\right)$. Taking 100 particle samples from $\mathcal{N}\left((0,0)^T,\, ((1,0),(0,1))^T\right)$ resulted in the plots shown in Figure 5. By removing $k(x_j, x)\nabla_{x_j}\log p(x_j)$, the particles were forced away (Figure 6). Similarly, removing $\nabla_{x_j}k(x_j, x)$ resulted in bunching of the particles (Figure 7).

References

1. Christian P Robert, George Casella, and George Casella. *Monte Carlo statistical methods*, volume 2. Springer, 1999.
2. Andrew Gelman, John B Carlin, Hal S Stern, and Donald B Rubin. Bayesian data analysis second edition corrected version (30 jan 2008). 1995.
3. Danilo Jimenez Rezende and Shakir Mohamed. Variational inference with normalizing flows, 2016.
4. Qiang Liu, Jason D. Lee, and Michael I. Jordan. A kernelized stein discrepancy for goodness-of-fit tests and model evaluation, 2016.
5. Alain Berlinet and Christine Thomas-Agnan. *Reproducing kernel Hilbert spaces in probability and statistics*. Springer Science & Business Media, 2011.
6. Erwin Kreyszig. *Introductory functional analysis with applications*, volume 17. John Wiley & Sons, 1991.
7. Ian Goodfellow et al. *Deep Learning*. MIT press, 2016.
8. Charles Stein. A bound for the error in the normal approximation to the distribution of a sum of dependent random variables. In *Proceedings of the Sixth Berkeley Symposium on Mathematical Statistics and Probability, Volume 2: Probability Theory*, volume 6, pages 583–603. University of California Press, 1972.
9. Jackson Gorham and Lester Mackey. Measuring sample quality with stein's method, 2019.
10. Qiang Liu and Dilin Wang. Stein variational gradient descent: A general purpose bayesian inference algorithm, 2019.
11. Qiang Liu. Stein variational gradient descent as gradient flow, 2017.
12. Qiang Liu and Dilin Wang. Stein variational gradient descent as moment matching, 2018.
13. Louis Sharrock and Christopher Nemeth. Coin sampling: Gradient-based bayesian inference without learning rates, 2023.

131 ## A Figures

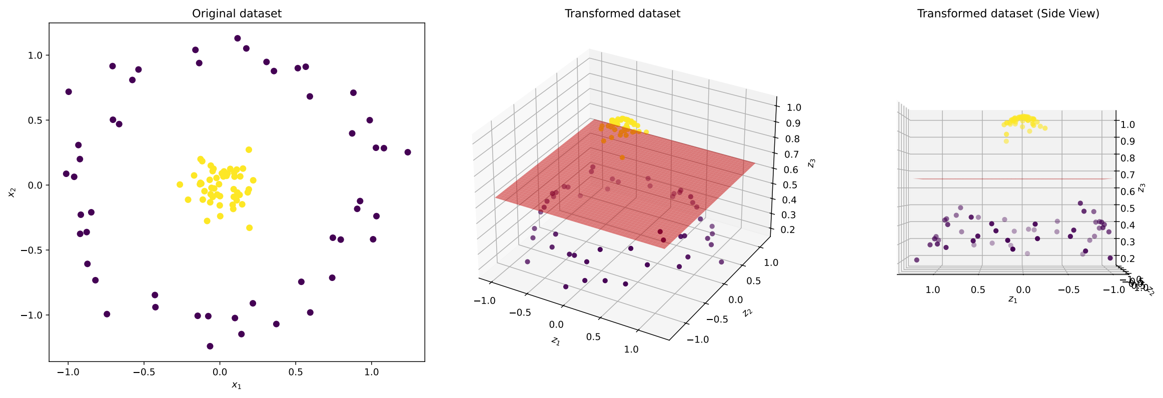

Figure 1: Plots from code 1, on the left we have the two two-dimensional input data which can not be categorize linearly. Two on the right demonstrated that after applying a suitable kernel function 1 the data can be categorize linearly in a higher dimension.

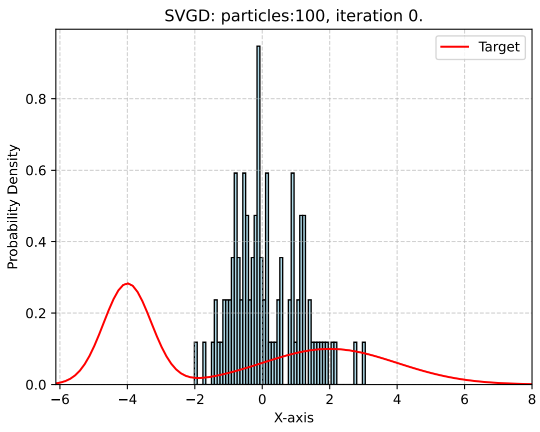

Figure 2: The histogram of the current sample of 100 particles with the red line being the target distribution. Here at iteration 0 we see that the histogram forms the initial sample distribution. From code 3.

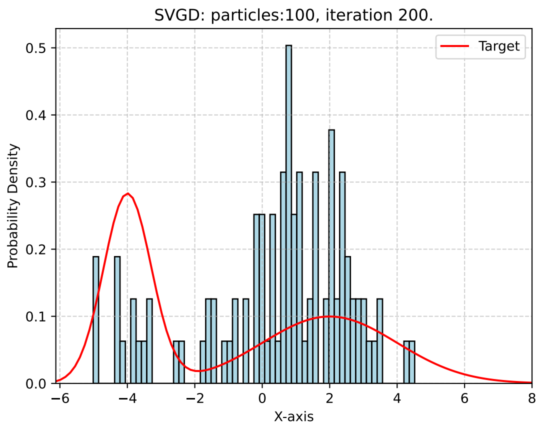

Figure 3: The histogram of the current sample of 100 particles with the red line being the target distribution.

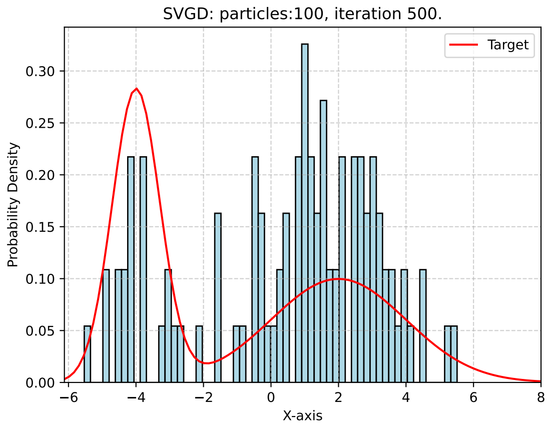

Figure 4: The histogram of the current sample of 100 particles with the red line being the target distribution.

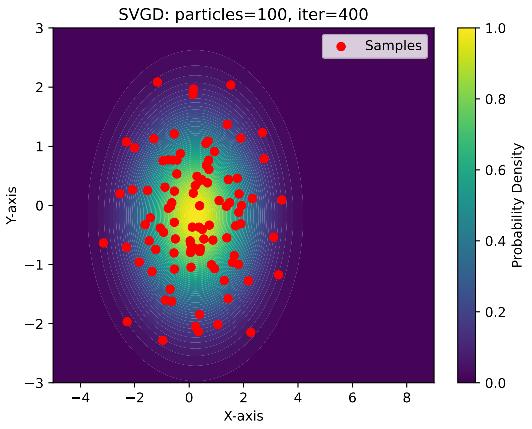

Figure 5: Here we overlay the 100 particles on the two-dimensional plot of a Gaussian after 400 iteration of our SVGD. From code 4.

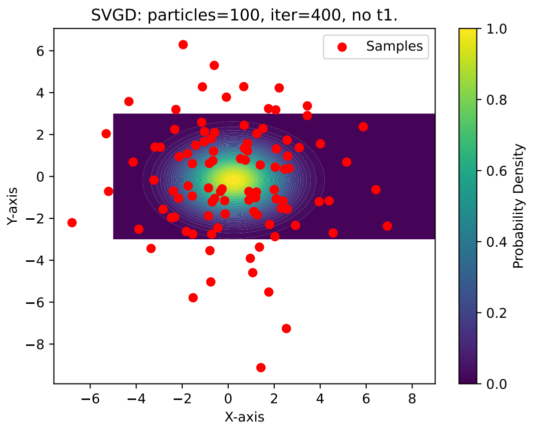

Figure 6: Removing the term that drives the particles towards the higher problems regions, we observed the spreading out as predict due to the repulsive force between each particles. Here we overlay the 100 particles on the two-dimensional plot of a Gaussian after 400 iteration of our SVGD. From code 4.

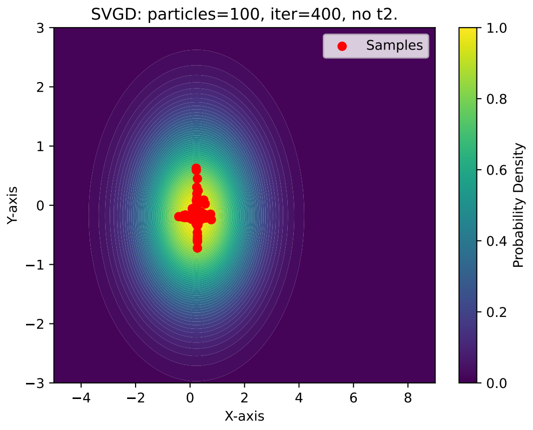

Figure 7: Removing the term that repulse each particle from each other leads to the bunching of particles at the higher region of probability. Here we overlay the 100 particles on the two-dimensional plot of a Gaussian after 400 iteration of our SVGD. From code 4.

B Algorithm

Algorithm 1 Bayesian Inference via Variational Gradient Descent. [10]

```
Require: A target distribution with density function $p(x)$ and a set of initial particles $\{x_i^0\}_{i=1}^n$
Ensure: A set of particles $\{x_i\}_{i=1}^n$ that approximates the target distribution.
1: for iteration = 1 to $l$ do
2:     for $i = 1$ to $n$ do
3:         $x_i' \leftarrow x_i + \epsilon\hat{\phi}^*(x_i)$                ▷ Update particles
4:         where $\hat{\phi}^*(x) = \frac{1}{n}\sum_{j=1}^n \left[k(x_j', x)\nabla_{x_j}\log p(x_j) + \nabla_{x_j}k(x_j', x)\right]$
5:     end for
6: end for
```

C Python Code

Listing 1: Kernel example.

```python
# Generate synthetic data
X, y = make_circles(n_samples=100, factor=.1, noise=.1)
Z = feature_map_2(X)

# 2D scatter plot
fig1 = plt.figure(figsize=(8, 8))
ax1 = fig1.add_subplot(1, 1, 1)
ax1.scatter(X[:, 0], X[:, 1], c=y, cmap='viridis')
ax1.set_xlabel('$x_1$')
ax1.set_ylabel('$x_2$')
ax1.set_title('Original␣dataset')

# Show the first plot
plt.show()

# 3D scatter plot
fig2 = plt.figure(figsize=(16, 8))
ax2 = fig2.add_subplot(1, 2, 1, projection='3d')
ax2.scatter3D(Z[:, 0], Z[:, 1], Z[:, 2], c=y, cmap='viridis')
ax2.set_xlabel('$z_1$')
ax2.set_ylabel('$z_2$')
ax2.set_zlabel('$z_3$')
ax2.set_title('Transformed␣dataset')

# Draw a hyperplane at z
z_hyperplane = 0.65
xx, yy = np.meshgrid(np.linspace(min(Z[:, 0]), max(Z[:, 0]), 50),
                     np.linspace(min(Z[:, 1]), max(Z[:, 1]), 50))
zz = np.full_like(xx, z_hyperplane)
ax2.plot_surface(xx, yy, zz, alpha=0.5, color='red', label='Hyperplane')

# Show the second plot
plt.show()

# 3D scatter plot (side view)
fig3 = plt.figure(figsize=(16, 8))
```

```
ax3 = fig3.add_subplot(1, 3, 1, projection='3d')
ax3.scatter3D(Z[:, 0], Z[:, 1], Z[:, 2], c=y, cmap='viridis')

# Draw a hyperplane at z=0.5
ax3.plot_surface(xx, yy, zz, alpha=0.5, color='red', label='Hyperplane')

ax3.set_xlabel('$z_1$')
ax3.set_ylabel('$z_2$')
ax3.set_zlabel('$z_3$')
ax3.set_title('Transformed␣dataset␣(Side␣View)')

# Set the view to the side
ax3.view_init(elev=0, azim=90)



# Adjust layout for better spacing
plt.tight_layout()
plt.show()
```

Listing 2: Simple SVGD implemented

```
import numpy as np
import matplotlib.pyplot as plt
import numpy.matlib as nm
def rbf_kernel(x, y, h=1):
    # Radial basis function (RBF) kernel
    return np.exp(-np.sum((x - y)**2) / (2 * h**2))

def compute_Kxy(x,h=1):
    # K xy
    # Dist function for all particles
    # dim(x) = N then we have a ( N x N) size matrix, with ones on the diag.

    N = len(x)
    Kxy = np.zeros((N, N))

    for i in range(N):
        for j in range(i):  # Only compute the lower half
            Kxy[i, j] = rbf_kernel(x[i], x[j], h)

    # Fill in the upper half using symmetry
    Kxy = Kxy + Kxy.T + np.diag(np.ones(N))

    return Kxy

def grad_kxy(x, Kxy, h=1):
    dx_kxy = -np.matmul(Kxy,x)
    sumkxy = np.sum(Kxy,axis=1)

    for i in range(x.shape[1]):
        dx_kxy[:,i] =  dx_kxy[:,i] + ( x[:,i] * sumkxy )
    #dx_kxy /= h**2

    return dx_kxy

def svgd_grad_X(x, log_prob_func ,):
    x = np.copy(x)
    # Computes phi^star for each time iteration.

    log_prob_grad = log_prob_func(x)

    # Compute the kernel matrix
    Kxy = compute_Kxy(x)
    dx_Kxy = grad_kxy(x,Kxy)

    term1 =  np.matmul(Kxy, log_prob_grad)
    term2 = dx_Kxy

    grad_x = (( term1 + term2) / x.shape[0])

    return grad_x


def svgd_run(x0, d_log_prob_func, n_iter=1, step_size=1e-3, alpha = 1):

    # adagrad with momentum
    fudge_factor = 1e-6

    x = np.copy(x0)
    x_hist = [np.ones_like(x)]*n_iter
    historical_grad =0
    #
    for n in range(n_iter):
        x_hist[n] = x
        grad_x = svgd_grad_X(x, d_log_prob_func)

        if n ==0:
            historical_grad = historical_grad + grad_x**2
        else:
            historical_grad = alpha * historical_grad + (1 - alpha) * (grad_x **

        # force
        F = np.divide(grad_x, fudge_factor+np.sqrt(historical_grad))

        # update
        x = x + step_size * F

    return x, x_hist
```

Listing 3: Bimodal Gaussian Example

```
# Bimodal
mu1 = -4
```

```
var1 = 0.5
mu2 = 2
var2 = 4

def bimodal_dlnprob(x):
    return (-1*(x-mu1)*(1/var1)*bimodal_gaussian(x, mu1, var1) - \
             (x-mu2)*(1/var2)*bimodal_gaussian(x, mu2, var2)) / \
             (bimodal_gaussian(x, mu1, var1) + bimodal_gaussian(x, mu2, var2))

x0 = np.random.normal(0,1, [100,1])

x, xh=svgd_run(x0, bimodal_dlnprob,500,0.01)

print("svgd: mu = {} var = {}".format(round(np.mean(x),2), round(np.std(x)**2,2)
plot_results(mu1, var1, mu2, var2, x)
```

Listing 4: 2D Unimodal Gaussian

```
# Unimodal Gaussian
A = np.array([[2,0],[0,1]])
mu = np.array([2,0])

def unimodal_gaussian_d_log_prob(x):
    return -1*np.matmul(x-nm.repmat(mu, x.shape[0], 1), np.linalg.inv(A))

np.random.seed(10)
x0 = np.random.normal(0,1, [50,2]);
xx, xh=svgd_run(x0, unimodal_gaussian_d_log_prob,500,0.01)
```