Answer 1

Assuming the norm is defined on the domain $\mathbb{D}$
Given $x, y \epsilon \mathbb{D}$, and $\lambda \epsilon \mathbb{R}^1, \lambda \geq 0$

$$f(\lambda x + (1 - \lambda)y) = ||\lambda x + (1 - \lambda)y||$$

$$f(\lambda x + (1 - \lambda)y) \leq ||\lambda x|| + ||(1 - \lambda)y||$$

$$f(\lambda x + (1 - \lambda)y) \leq \lambda ||x|| + (1 - \lambda)||y||$$

$$f(\lambda x + (1 - \lambda)y) \leq \lambda f(x) + (1 - \lambda)f(y)$$

This means the function is convex.

Answer2

Consider the probability bound for any of the samples lying in the set $\mathcal{S}$.
Focus on the j^{th} dimension of x and x_0,
If $x_{0,j} \,\epsilon\, [-1,1]$

$$P(\|x_{i,j} - x_{0,j}\|_\infty \leq 0.99) = \frac{\text{length of the real number line between [-1,1] which is within 0.99 distance of} x_{0,j}}{\text{length of the real number line between [-1,1]}}$$

$$P(\|x_{i,j} - x_{0,j}\|_\infty \leq 0.99) = \begin{cases} \frac{2*0.99}{2} = 0.99 & \text{if } x_{0,j} < 0.01 \\ \frac{0.99+(1-|x_{0,j}|)}{2} \leq 0.99 & \text{if } x_{0,j} \geq 0.01 \end{cases}$$

If $x_{0,j} \,\epsilon\, (-\infty, -1)U(1,\infty)$

$$P(\|x_{i,j} - x_{0,j}\|_\infty \leq 0.99) = \begin{cases} \frac{0}{2} = 0 & \text{if } |x_{0,j}| > 1.99 \\ \frac{|x_{0,j}|-1)}{2} < 0.495 & \text{if } |x_{0,j}| \leq 1.99 \end{cases}$$

This means,

$$P(\|x_{i,j} - x_{0,j}\|_\infty \leq 0.99) \leq 0.99$$

Now for the d-dimensional case, for any sample $x_i \epsilon\, \mathcal{Q}$ to also $\epsilon\, \mathcal{S}$, $\|x_{i,j} - x_{0,j}\|_\infty \leq 0.99$ should hold $\forall j \epsilon [d]$. Now each dimension being independent of the other we can write,

$$P(\|x_i - x_0\|_\infty \leq 0.99) \leq (0.99)^d$$

Now for all the N samples to hold this true, given they are all sampled with uniform probability i.e. the samples are independent of each other

$$P(\|x_i - x_0\|_\infty \leq 0.99 \forall i \epsilon [N]) \leq (0.99)^{dN}$$

Now,

$$P((\|x_i - x_0\|_\infty \leq 0.99 \forall i \epsilon [N])^c) \geq 1.0 - (0.99)^{dN}$$

Also the event

$$(\|x_i - x_0\|_\infty \leq 0.99 \forall i \epsilon [N])^c) \subset (\|x_i - x_0\|_\infty > 0.99)$$

This means,

$$P(\|x_i - x_0\|_\infty > 0.99 \forall i \epsilon [N]) \geq P((\|x_i - x_0\|_\infty \leq 0.99 \forall i \epsilon [N])^c) \geq (1.0 - (0.99)^{dN})$$

Now given that $N = d^{O(1)}$, that means $dN = d^{O(1)+1}$

$$P(\|x_i - x_0\|_\infty > 0.99 \forall i \epsilon [N]) \geq (1.0 - (0.99)^{d^{O(1)+1}}) \geq (1.0 - (0.99)^d)$$

Now one can write $(0.99)^d = \exp(d * \ln 0.99)$

$$P(\|x_i - x_0\|_\infty > 0.99 \forall i \epsilon [N]) \geq (1.0 - \exp(d * \ln 0.99)) \geq (1.0 - \exp(-d))$$

$$P(\|x_i - x_0\|_\infty > 0.99 \forall i \epsilon [N]) \geq (1.0 - \exp(-\Omega(d)))$$

For the optimization algorithm to work we need to sample a point within δ neighborhood of $x^{*} \epsilon D$ The above bound can be written for any δ as

$$P(\|x_i - x_0\|_\infty > \delta \forall i \epsilon [N])) \geq (1.0 - (delta)^d)$$

This means the probability of atleast one point lying in the δ range is;

$$P((\|x_i - x_0\|_\infty > \delta \forall i \epsilon [N]))^c) \leq (delta)^d$$

Clearly this probability decreases exponentially in d, so it is not possible to perform the optimization using the given algorith with high probability by selecting $O(d)$ points.

Answer3.1

$$f(x_t) = f(x_{t-1} - \eta\widetilde{\nabla}f(x_{t-1})) = (x_{t-1} - \eta\widetilde{\nabla}f(x_{t-1}))^2$$

$$f(x_t) = x_{t-1}^2 + \eta^2(\widetilde{\nabla}f(x_{t-1}))^2 - 2\eta\widetilde{\nabla}f(x_{t-1})x_{t-1}$$

$$f(x_t) = x_{t-1}^2 + \eta^2(\nabla f(x_{t-1}))^2(1 + (\xi)^2 + 2\xi) - 2\eta\nabla f(x_{t-1})(1+\xi)x_{t-1}$$

Also $\nabla f(x) = 2 * x$

$$f(x_t) = x_{t-1}^2(1 + 4\eta^2(1 + (\xi)^2 + 2\xi) - 4\eta(1+\xi))$$

$$f(x_t) = f(x_{t-1})(1 + 4\eta^2(1 + (\xi)^2 + 2\xi) - 4\eta(1+\xi))$$

Now for a given x_{t-1},

$$\mathbb{E}[f(x_t)] = f(x_{t-1})\,\mathbb{E}[(1 + 4\eta^2(1 + (\xi)^2 + 2\xi) - 4\eta(1+\xi))]$$

Now $f(x) > 0 \,\forall\, x$, so for $\mathbb{E}[f(x_{t+1}] \to \infty$,

$$\mathbb{E}[(1 + 4\eta^2(1 + (\xi)^2 + 2\xi) - 4\eta(1+\xi))] \geq 1$$

$$1 + 4\eta^2(1 + \sigma^2) - 4\eta \geq 1$$

Given $\eta >= 0$

$$\eta \geq \frac{1.0}{1.0 + \sigma^2} \geq \frac{2}{\sigma}$$

3.2.1

Using given values,

$$x_{t+1} = x_t - 2g_t$$

$$g_t = 0.9g_{t-1} + 0.2x_t$$

Using g_t in the update rule for x_{t+1}

$$x_{t+1} = x_t - 2(0.9g_{t-1} + 0.2x_t) = 0.6x_t - 1.8g_{t-1}$$

Also,

$$x_t = x_{t-1} - 2g_{t-1}$$

$$g_{t-1} = 0.5x_{t-1} - 0.5x_t$$

Using this we can write,

$$x_{t+1} = 1.5x_t - 0.9x_{t-1}$$

Combining with,

$$x_t = x_t + 0x_{t-1}$$

$$\begin{bmatrix} x_{t+1} \\ x_t \end{bmatrix} = \begin{bmatrix} 1.5 & -0.9 \\ 1.0 & -0 \end{bmatrix} \begin{bmatrix} x_t \\ x_{t-1} \end{bmatrix}$$

This means,

$$M = \begin{bmatrix} 1.5 & -0.9 \\ 1.0 & -0 \end{bmatrix}$$

$$\begin{bmatrix} x_{t+1} \\ x_t \end{bmatrix} = U \begin{bmatrix} 0.75000 + 0.58095i & -0.9 \\ 1.0 & -0.75000 - 0.58095i \end{bmatrix} U^T \begin{bmatrix} x_t \\ x_{t-1} \end{bmatrix}$$

$$\begin{bmatrix} x_{t+1} \\ x_t \end{bmatrix} = U \begin{bmatrix} (0.75000 + 0.58095i)^t & 0 \\ 0 & -(0.75000 - 0.58095i)^t \end{bmatrix}^t U^T \begin{bmatrix} x_1 \\ x_0 \end{bmatrix}$$

The eigen values of M are: $0.75000 + 0.58095i, 0.75000 - 0.58095i$, both have a norm of 0.948, and hence the values of x_t will decay exponentially with t, i.e. $x_t = \exp(-\Omega(t))x_0$.
Now $f(x_t) = (x_t)^2 = \exp(-2\Omega(t))(x_0)^2 = \exp(-\Omega(t))f(x_0)$

Answer 3.2.2

$$dh(t) = h(t+1) - h(t) = x_{t+1} - x_t = -\eta g_t$$

$$\frac{d^2h(t)}{d\tau(t)^2} = \frac{dh(t+1) - dh(t)}{(\tau(t+1) - \tau(t))^2} = -\eta(g_{t+1} - g_t)m^2$$

$$g_{t+1} = (1-\gamma)g_t + \gamma\nabla f(x_{t+1}) = (1-\gamma)g_t + 2a\gamma x_{t+1}$$

Using this, we can write

$$\frac{d^2h(t)}{d\tau(t)^2} = (\eta_0\gamma_0 g_t - \eta_0\gamma_0 a x_{t+1})$$

Also,

$$\frac{dh(t)}{d\tau(t)} = -\eta g_t m = -\eta_0 g_t$$

$$\frac{d^2h(t)}{d\tau(t)^2} = -\gamma_0\frac{dh(t)}{d\tau(t)} - 2\eta_0\gamma_0 a x_{t+1}$$

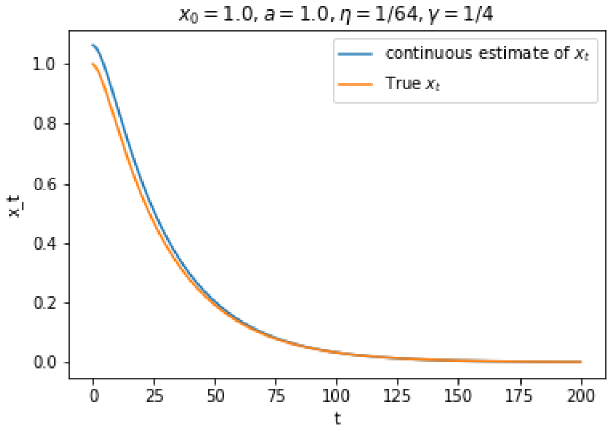
$$x_{t+1} = x_t - \eta g_t = h(t) - \frac{\eta_0}{m}g_t = h(t) - \frac{1}{m}\frac{dh(t)}{d\tau(t)}$$

$$\frac{d^2h(t)}{d\tau(t)^2} = -\gamma_0\frac{dh(t)}{d\tau(t)} - 2\eta_0\gamma_0 a h(t) - 2\eta_0\gamma_0 a\frac{dh(t)}{d\tau(t)}$$

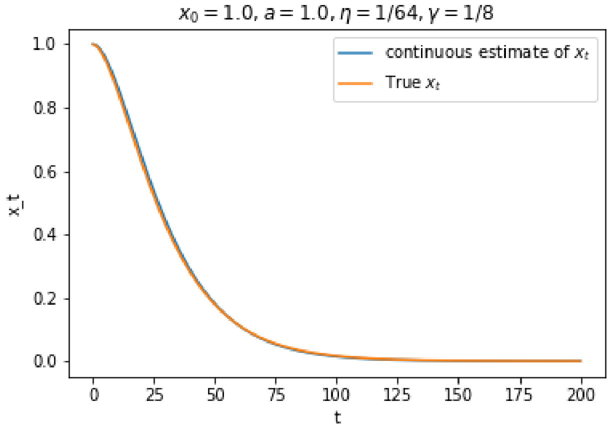
This means,
$c_1(a, \eta_0, \gamma_0) = -\gamma_0$, $c_2(a, \eta_0, \gamma_0) = -2\eta_0\gamma_0 a$, and $\epsilon = -2\eta_0\gamma_0 a\frac{dh(t)}{d\tau(t)}$

Answer 3.2.3

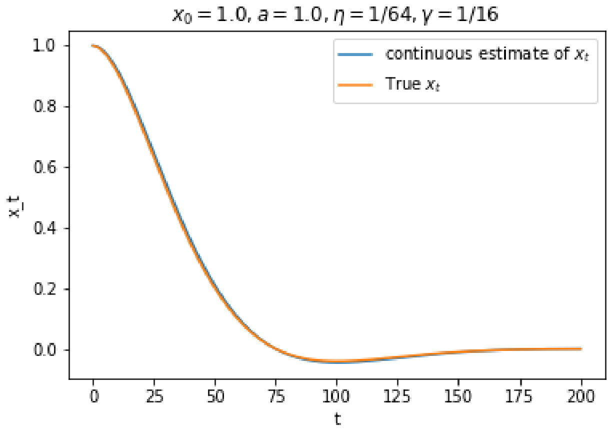
- $x_0 = 1, a = 1, \eta = \frac{1}{64}, \gamma = \frac{1}{4}$
  Solution: $h(t) = 1.270711 \exp{(-0.0366117t)} - 0.207107 \exp{(-0.213388t)}$



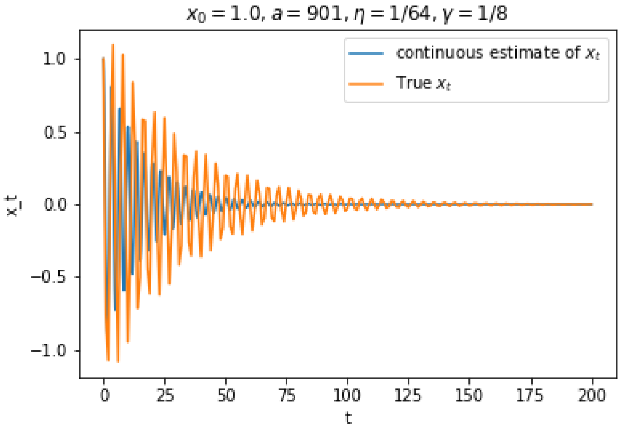
- $x_0 = 1, a = 1, \eta = \frac{1}{64}, \gamma = \frac{1}{8}$
  Solution: $h(t) = \frac{1}{16} \exp{(-\frac{t}{16})} + (t + 16)$



- $x_0 = 1, a = 1, \eta = \frac{1}{64}, \gamma = \frac{1}{16}$
  Solution: $h(t) = \exp{(\frac{-t}{32})}(\sin{(\frac{t}{32}} + \cos{(\frac{t}{32})}$



- $x_0 = 1, a = 901, \eta = \frac{1}{64}, \gamma = \frac{1}{8}$
  Solution: $h(t) = \frac{1}{30} \exp\left(\frac{-t}{16}\right)(\sin\left(\frac{15t}{8}\right) + 30\cos\left(\frac{15t}{8}\right)$



The estimate breaks down when the value of a is high, as this leads to a high value of $\nabla f(x_{t+1})$, and this the effect of past gradients also gets scaled which leads to very bad steps being taken, and hence ther is a wild oscillation that can be seen in the last plot.
For a given a, η, for smaller γ the algorithm follows the past gradients more than the current one, due to this after reaching minima where current gradient is close to 0, the algorithm stills takes bigger steps due to the past gradients, and this stabilizes slowly over there. If the value of γ is kept high, the algorithm will stabilize quicker when it reaches the minima.

Answer 4

Gradient of $f(x)$ can be written as:

$$\nabla f(x) = [2ax_1, 2x_2]^T$$

Given the partial derivative of $f(x)$ w.r.t x_i is only a function of x_i, the update rule for each coordinate can be written separetely without considering the update rule for the other.

$$[x_t]_1 = [x_{t-1}]_1 - \frac{2a[x_{t-1}]_1}{\sqrt{\sum_{s\leq(t-1)}(2a[x_s]_1)}}$$

$$[x_t]_1 = [x_{t-1}]_1 - \frac{[x_{t-1}]_1}{\sqrt{\sum_{s\leq(t-1)}([x_s]_1)}}$$

$$[x_t]_2 = [x_{t-1}]_2 - \frac{[x_{t-1}]_2}{\sqrt{\sum_{s\leq(t-1)}([x_s]_2)}}$$

Clearly both the coordinates have the same functional form of the update rule (independent of a), and given that they both start at the same starting point $[x_0]_i = c,\ i \,\epsilon\, [1, 2]$, this means both the coordinates will stay the same $\forall\, t$, i.e.

$$[x_t]_1 = [x_t]_2$$

Now we can write γ as:

$$\gamma = \sqrt{\frac{\sum_{s\leq t}[\nabla f(x_s)]_1^2}{\sum_{s\leq t}[\nabla f(x_s)]_2^2}}$$

$$\gamma = \sqrt{\frac{(2a)^2\sum_{s\leq t}[x_s]_1^2}{2^2\sum_{s\leq t}[x_s]_2^2}}$$

Now given

$$[x_s]_1 = [x_s]_2\ \forall s$$

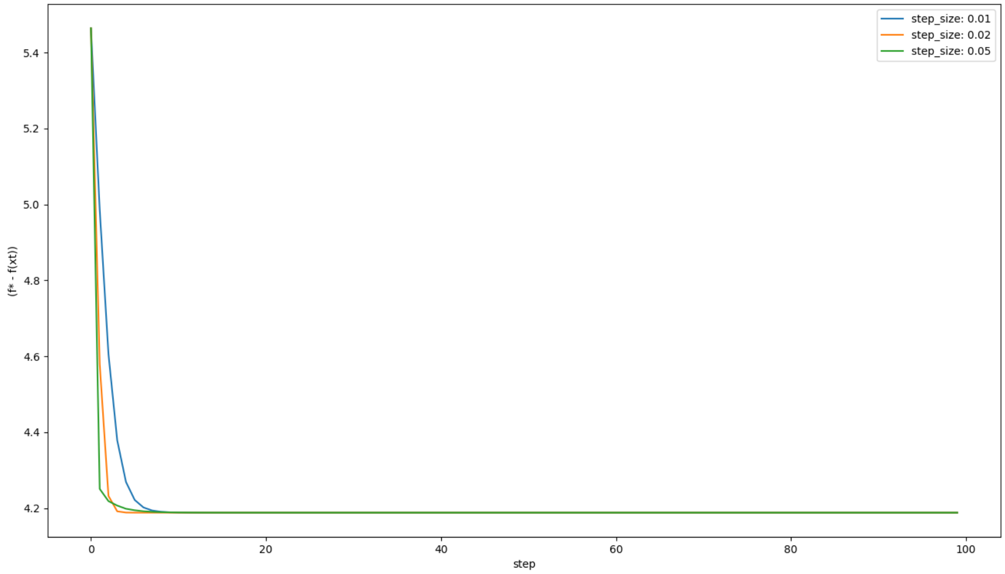
$$\sum_{s\leq t}[x_s]_1^2 = \sum_{s\leq t}[x_s]_2^2$$

Thus we can write γ as:

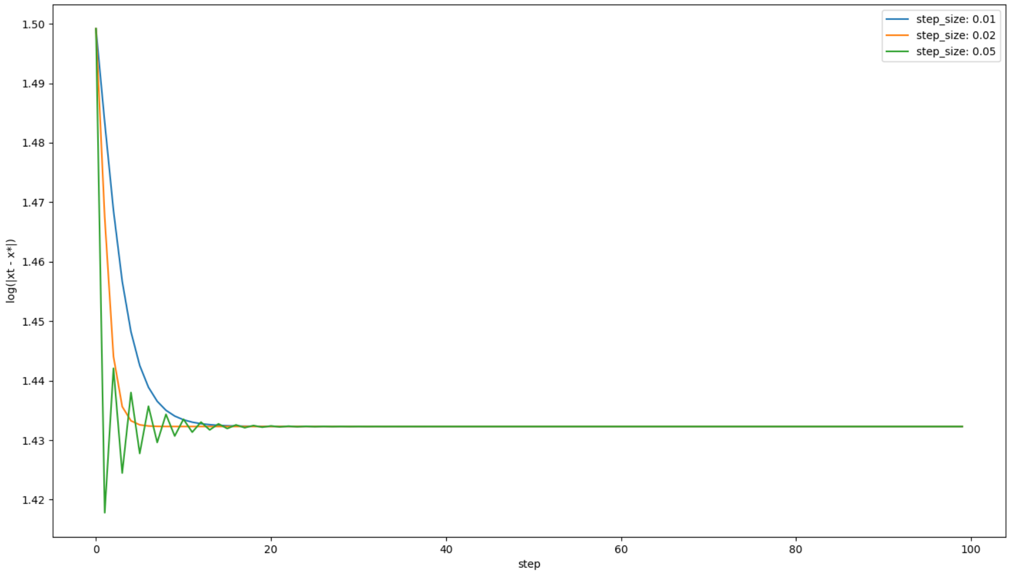
$$\gamma = a$$

Answer 5.1

$f^\star - f(x_t)$



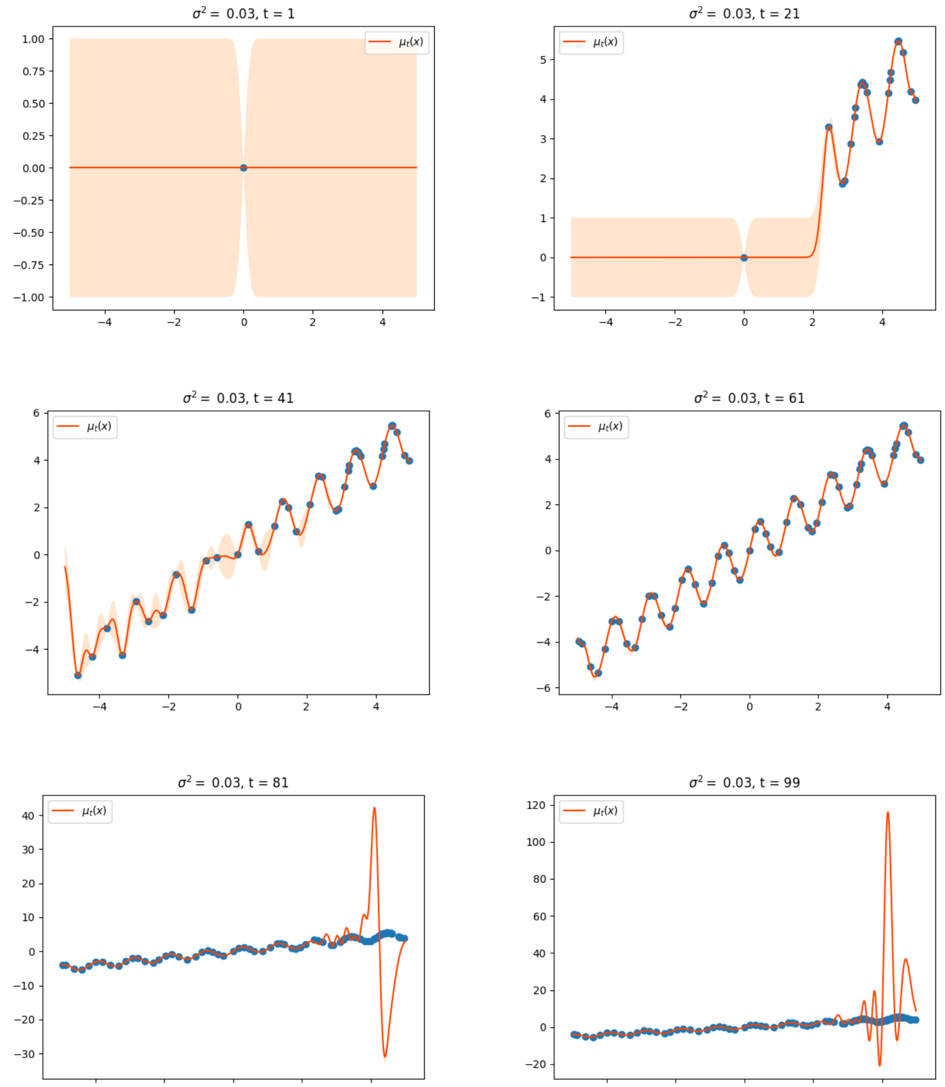
$|x_t - x^\star|$



As the step size is increased, the algorithm reaches the optimum value faster, but for the highest step size of 0.05, the algorithm starts to oscillate around the minima, but eventually decays to the same minima as the others.

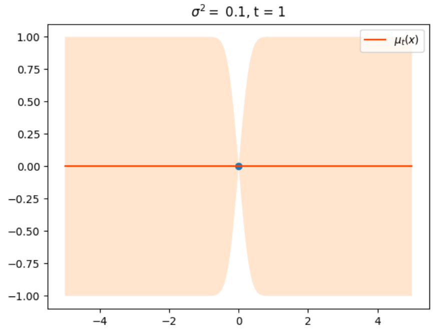
Answer 5.2

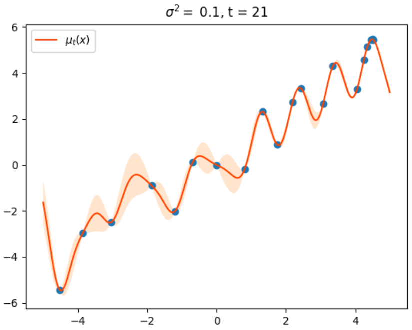
- $\sigma^2 = 0.03$

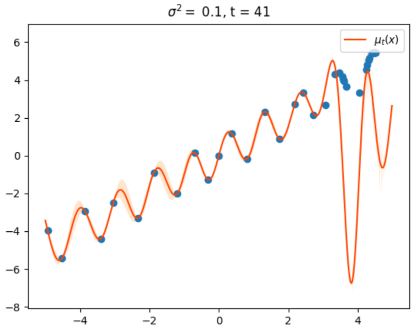


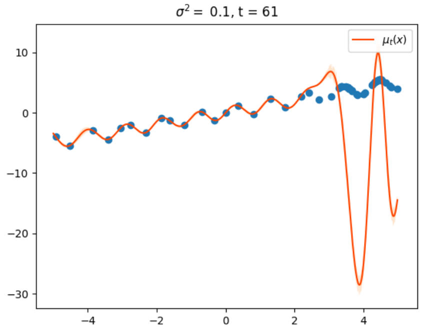
Final value: (x,f) = (4.4782464, 5.46450978)

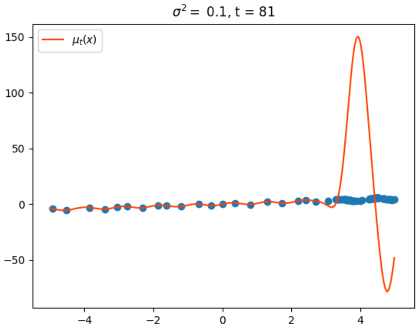
- $\sigma^2 = 0.1$

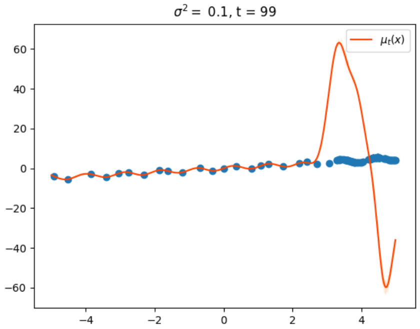






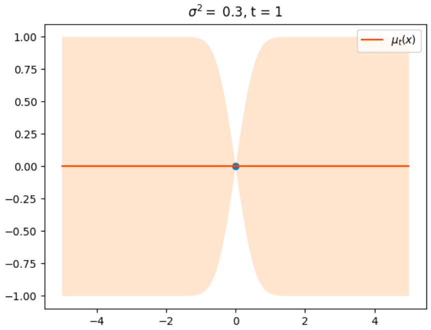


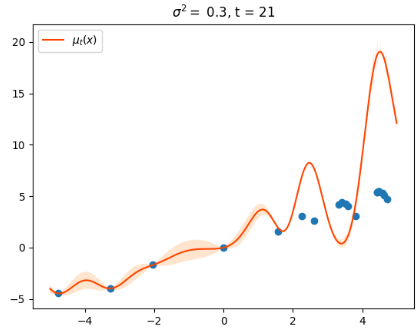


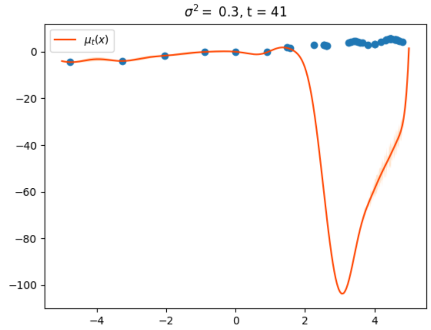


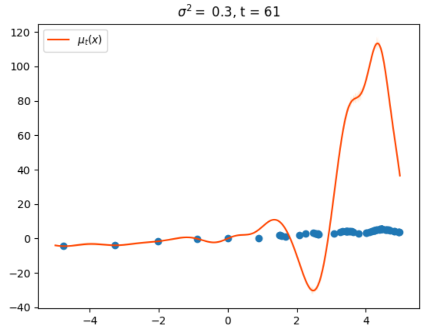
Final value: (x,f) = (4.48194939, 5.46429969)

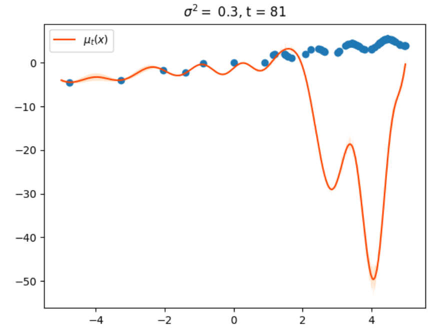
- $\sigma^2 = 0.3$

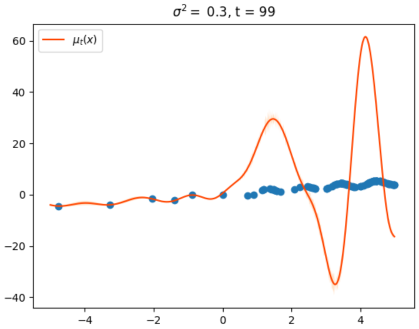






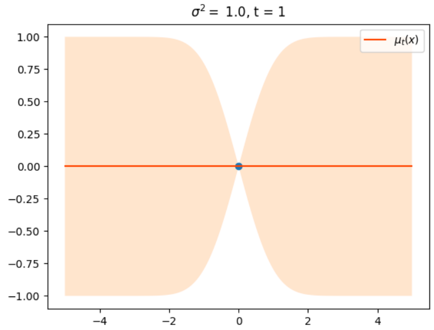


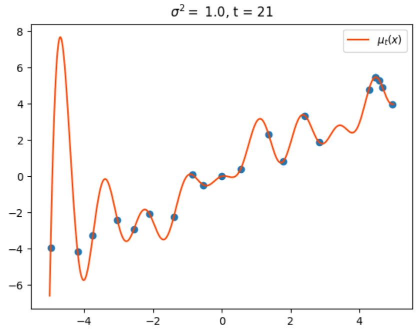


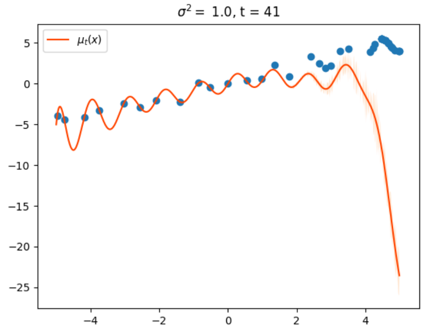


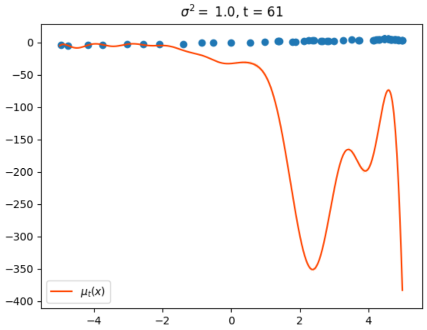
Final value: (x,f) = (4.47029106, 5.46331253)

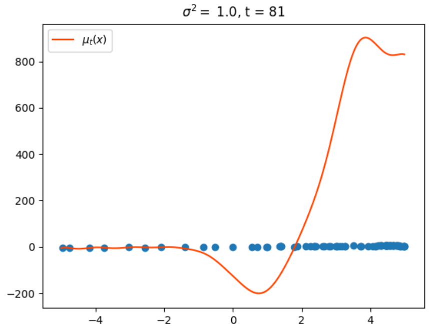
- $\sigma^2 = 1.0$

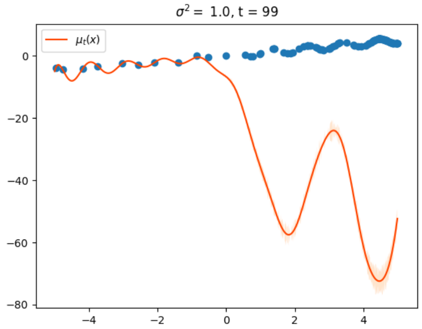






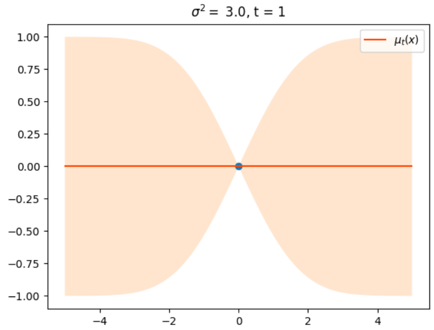


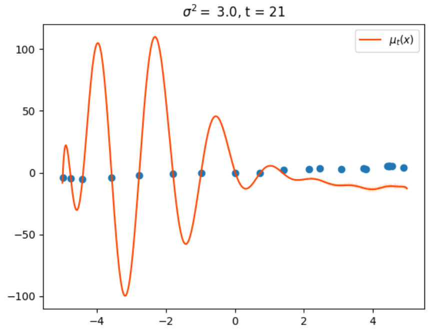


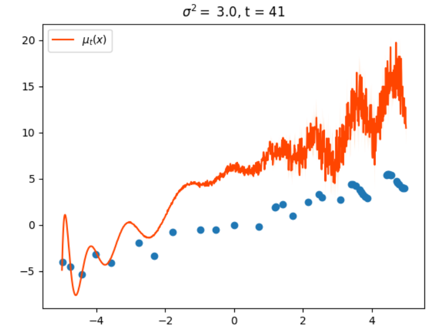


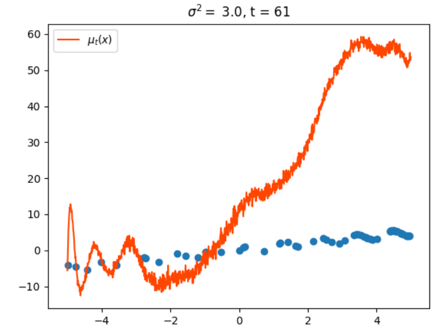
Final value: (x,f) = (4.4790136051787535, 5.4645061782361495)

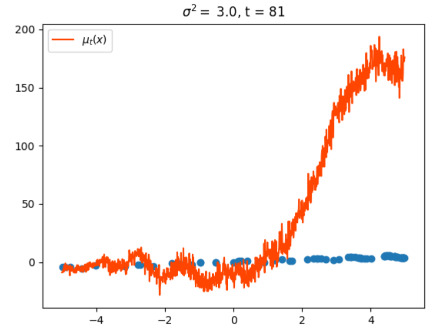
- $\sigma^2 = 3$

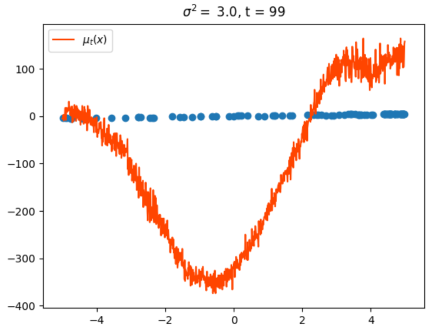






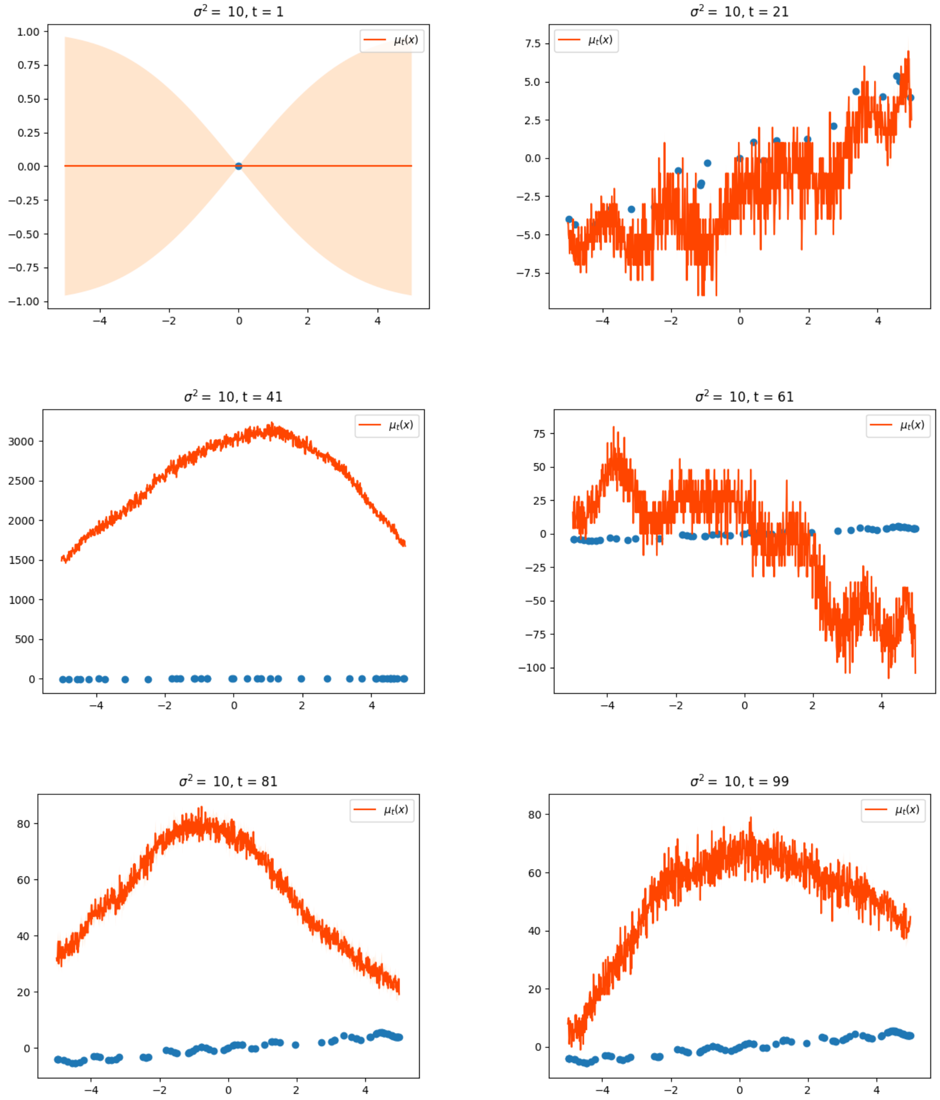






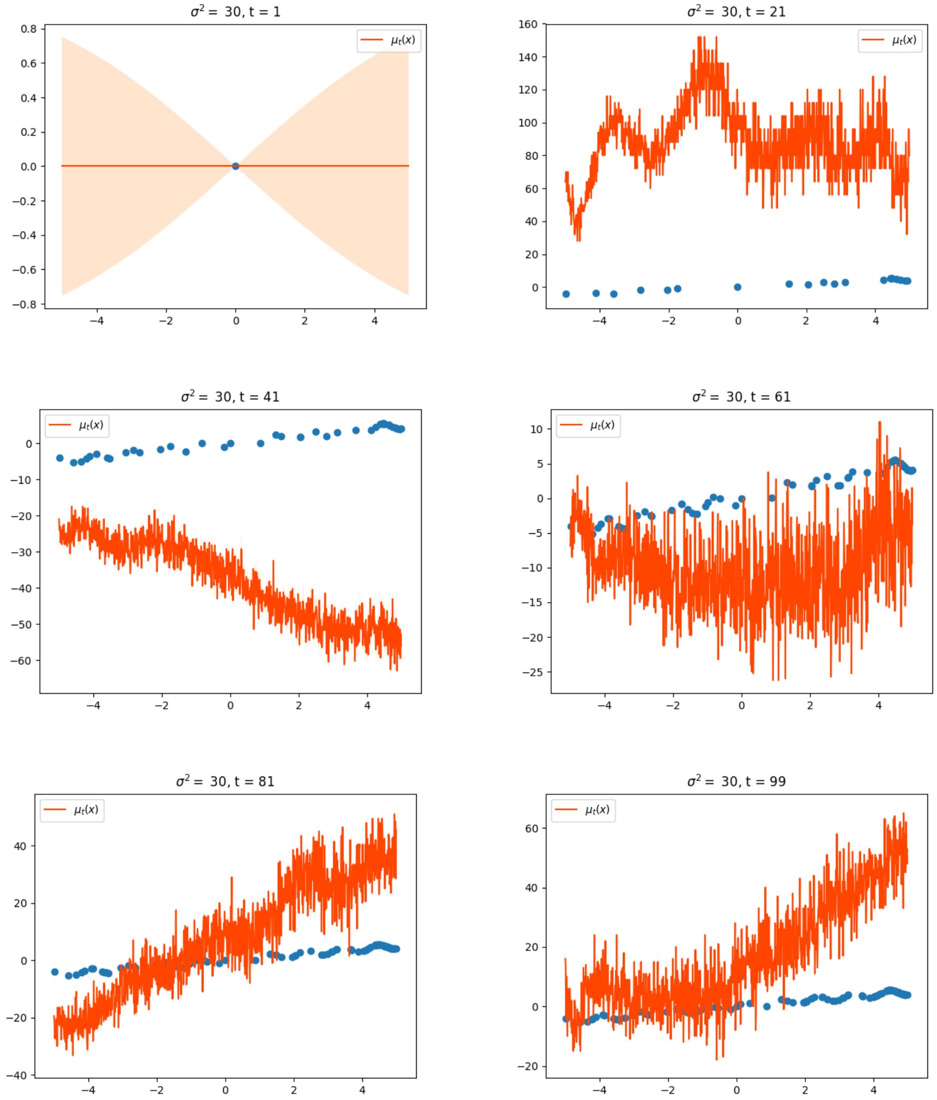
Final value: (x,f) = (4.48194939, 5.46429969)

- $\sigma^2 = 10$



Final value: (x,f) = (4.470956012680244, 5.463499059630465)

- $\sigma^2 = 30$

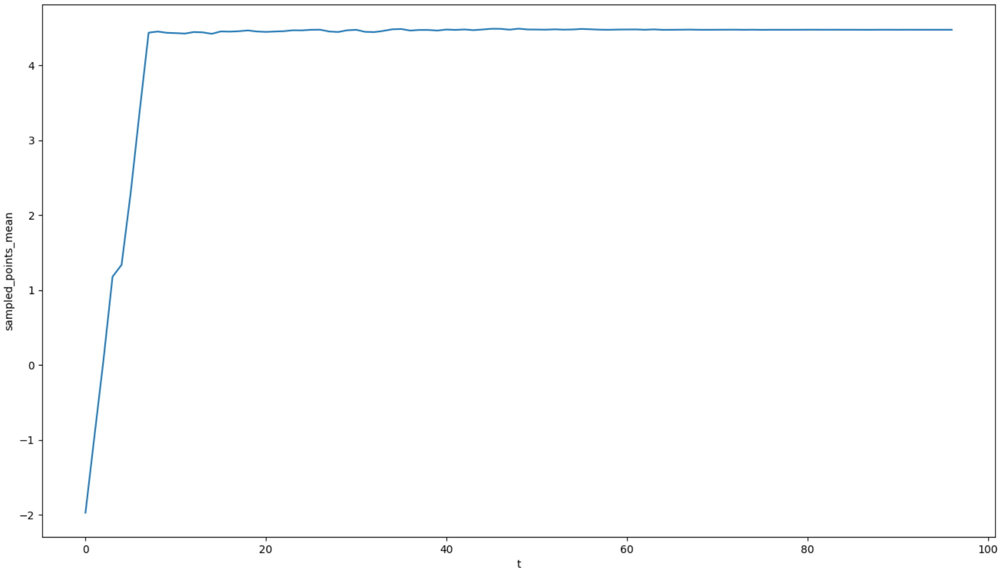


Final value: (x,f) = (4.481363086251285, 5.464365317322305)

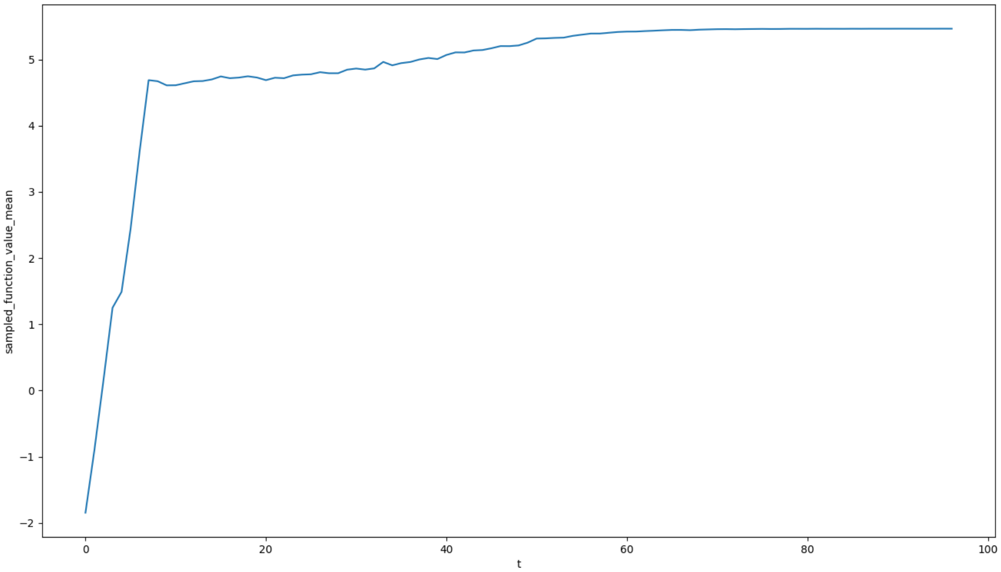
For all the sigma values, the optimization algorithm is able to find the optima value, but it seems to fit the function, only for the smaller sigma values. It could be because for higher sigma, the kernel gives the same inverse distance for all the points, which might be making it difficult to find the perfect fit.

Answer 5.3

$\frac{1}{T}\sum_{s\in[T]} x_s^{(t)}$



$\frac{1}{T}\sum_{s\in[T]} f(x_s^{(t)})$



I used $T = 500$ for the Metropolis-Hasting algorithm The plots clearly show a very fast convergence to the optima value. Though one thing I noticed was that if the value of T is kept low, the samples produced by the Metropolis-Hasting algorithm are not very good, and lead to very noisy updates initially.

q32

April 19, 2020

[ ]:

```
import numpy as np
import matplotlib.pyplot as plt
```

[ ]:

```
def h(t,a,b,c,d):
    return a*np.exp(b*t) + c*np.exp(d*t)
```

[ ]:

```
def momentum_update(x0, A, eta, gamma, num_iters=200):
    X = [x0*1.0]
    x = x0*1.0
    g = 0.0
    for t in range(num_iters):
        g = (1-gamma)*g + gamma*(2*A*x)
        x = x - eta*g
#         print(x)
        X.append(x*1.0)
    return X

```

[ ]:

```
a = 1.270711
b = -0.036617
c = -0.207107
d = -0.213388
x0 = 1.0
A = 1.0
eta = 1/64
gamma = 1/4
num_iters = 200
def h(t):
    return a*np.exp(b*t) + c*np.exp(d*t)

t = np.arange(0,200,0.01)
H = h(t)
X = momentum_update(x0, A, eta, gamma, num_iters)
plt.plot(t,H,label=r'continuous estimate of $x_t$')
plt.plot(X, label=r'True $x_t$')
plt.xlabel("t")
plt.ylabel(r'x_t')
plt.title(r'$x_0 = {}, a = {}, \eta = 1/{}, \gamma = 1/{}$'.format(x0,A,64, 4))
plt.legend()
plt.savefig("./q32_1.png")
plt.show()
```

[ ]:

```
a = 1.270711
b = -0.036617
c = -0.207107
d = -0.213388
x0 = 1.0
A = 1.0
eta = 1/64
gamma = 1/8
num_iters = 200
def h(t):
    return (1/16)*(np.exp(-t/16))*(t+16)

t = np.arange(0,200,0.01)
H = h(t)
X = momentum_update(x0, A, eta, gamma, num_iters)
plt.plot(t,H,label=r'continuous estimate of $x_t$')
plt.plot(X, label=r'True $x_t$')
plt.xlabel("t")
plt.ylabel(r'x_t')
plt.title(r'$x_0 = {}, a = {}, \eta = 1/{}, \gamma = 1/{}$'.format(x0,A,64, 8))
plt.legend()
plt.savefig("./q32_2.png")
plt.show()
```

[ ]:

```
a = 1.270711
b = -0.036617
c = -0.207107
d = -0.213388
x0 = 1.0
A = 1.0
eta = 1/64
gamma = 1/16
num_iters = 200
def h(t):
    return np.exp(-t/32)*(np.sin(t/32) + np.cos(t/32))

t = np.arange(0,200,0.01)
H = h(t)
X = momentum_update(x0, A, eta, gamma, num_iters)
plt.plot(t,H,label=r'continuous estimate of $x_t$')
plt.plot(X, label=r'True $x_t$')
plt.xlabel("t")
plt.ylabel(r'x_t')
plt.title(r'$x_0 = {}, a = {}, \eta = 1/{}, \gamma = 1/{}$'.format(x0,A,64, 16))
plt.legend()
plt.savefig("./q32_3.png")
plt.show()
```

[ ]:

```
a = 1.270711
b = -0.036617
c = -0.207107
d = -0.213388
x0 = 1.0
A = 901
eta = 1/64
gamma = 1/16
num_iters = 200
def h(t):
    return (1/30)*np.exp(-t/16)*(np.sin(15*t/8) + 30*np.cos(15*t/8))

t = np.arange(0,200,0.01)
H = h(t)
X = momentum_update(x0, A, eta, gamma, num_iters)
plt.plot(t,H,label=r'continuous estimate of $x_t$')
plt.plot(X, label=r'True $x_t$')
plt.xlabel("t")
plt.ylabel(r'x_t')
plt.title(r'$x_0 = {}, a = {}, \eta = 1/{}, \gamma = 1/{}$'.format(x0,A,64, 8))
plt.legend()
plt.savefig("./q32_4.png")
plt.show()
```

```python
import numpy as np
from matplotlib import pyplot as plt
import argparse


def f(x):
    val = x + np.sin(6*x)
    return val

def grad_f(x):

    grad = 1.0 + 6.0*np.cos(6*x)
    return grad

def grad_ascent(func, grad_func, x0, lr, num_steps):

    f_max = 5.464
    x_max = 4.478

    x_arr = np.zeros(shape=(num_steps,))
    f_arr = np.zeros(shape=(num_steps,))

    x = x0*1.0
    x_arr[0] = x*1.0
    f_arr[0] = (func(x))

    for i in range(1,num_steps):

        grad_f = grad_func(x)

        x = x + lr*grad_f
        x_arr[i] = x*1.0
        f_arr[i] = func(x)

    f_arr = f_max - f_arr
    x_arr = np.abs(x_arr - x_max)
    return f_arr, np.log(x_arr)


def plot_f(data_arr, lr_arr, ylabel=None, figname=None):
    fig = plt.figure(figsize=(16,9))

    for data, lr in zip(data_arr, lr_arr):
        plt.plot(data, label="step_size: {}".format(lr))
    plt.xlabel("step")
    if(ylabel):
        plt.ylabel(ylabel)
    plt.legend()
    plt.savefig("q51_{}.png".format(figname))
    plt.close()

def main():
    x0 = 0.0
    lrs = [0.01, 0.02, 0.05]
    f_arr = []
    x_arr = []
    num_steps = 100
    for lr in lrs:
        f_vals, xs = grad_ascent(f, grad_f, x0, lr, num_steps)
        f_arr.append(f_vals)
        x_arr.append(xs)

    plot_f(f_arr, lrs, ylabel="(f* - f(xt))", figname="f_vals")
    plot_f(x_arr, lrs, ylabel="log(|xt - x*|)", figname="x_vals")

if __name__ == "__main__":
    main()
```

```
import numpy as np
from matplotlib import pyplot as plt
import argparse
from q5_1 import *
import ipdb
from scipy.stats import norm
import math
import os
class GPR():
    def __init__(self, var_K):

        self.var_K = var_K
        self.F = None
        self.M = None
        self.M_inv = None
        self.X = None

    def gaussian_kernel(self,x,y):
        return np.exp(-0.5*np.linalg.norm(x-y, ord=2, axis=-1)**2 / self.var_K)

    def objective_func(self,x):
        return x

    def fit(self, X,F):
        XX = np.vstack([X[:,0]]*X.shape[0])
        XXT = np.copy(XX).T.reshape(-1,1)
        XX = XX.reshape(-1,1)

        # M = np.zeros(shape=(X.shape[0],X.shape[0]))
        # for i,x in enumerate(X[:,0]):
            # M[i,:] = self.gaussian_kernel(x*np.ones_like(X), X)

        # ipdb.set_trace()
        M = self.gaussian_kernel(XXT,XX).reshape(X.shape[0],X.shape[0])
        self.M_inv = np.linalg.inv(M+1e-7)
        self.M = M
        self.F = F
        self.X = X

    def predict(self,x):
        v = self.gaussian_kernel(x*np.ones_like(self.X), self.X)
        x_arr = np.array(x).reshape(-1,1)

        # mu = np.matmul(v.T, np.linalg.solve(self.M, self.F))
        # var = self.gaussian_kernel(x_arr,x_arr) - np.matmul(v.T, np.linalg.solve(sel
f.M, v)) + 1e-4

        mu = np.matmul(v.T, np.matmul(self.M_inv, self.F))
        var = self.gaussian_kernel(x_arr,x_arr) - np.matmul(v.T, np.matmul(self.M_inv,
 v))
        if(var<0):
            var = 0.0
        self.mu = mu
        self.var = var
        sample_vals = np.random.normal(mu, var**0.5, 100)

        id_max = np.argmax(sample_vals)
        return mu
        # return sample_vals[id_max]

    def get_mean_std(self):
        X = np.arange(-5,5,0.01)
        means = []
        stds = []
        for x in X:
            v = self.gaussian_kernel(x*np.ones_like(self.X), self.X)
            x_arr = np.array(x).reshape(-1,1)

            # mu = np.matmul(v.T, np.linalg.solve(self.M, self.F))
            # var = self.gaussian_kernel(x_arr,x_arr) - np.matmul(v.T, np.linalg.solve
(self.M, v)) + 1e-4

            mu = np.matmul(v.T, np.matmul(self.M_inv, self.F))
            var = self.gaussian_kernel(x_arr,x_arr) - np.matmul(v.T, np.matmul(self.M_
inv, v))
            if(var<0):
                var = np.array(0).reshape(var.shape)
            means.append(mu[0]*1.0)
            stds.append(var[0]**0.5)
        return X, np.array(means), np.array(stds)

    def acquisition_func(self, x, ft_max):
        _ = self.predict(x)
        phi = norm.cdf((self.mu-ft_max)/self.var**0.5)
        ei = (self.mu - ft_max) * phi + (self.var**0.5 / (np.sqrt(2*np.pi))) * np.exp(
-(ft_max - self.mu)**2 / (2*self.var))

        return ei

    def optimize_acq_func(self, ft_max):

        X = np.random.uniform(-5,5, 100)
        Y = np.array([self.acquisition_func(x, ft_max) for x in X])

        return X[np.argmax(Y)]



def bayes_optim(model, X, Y, num_iters, dir_name):

    for i in range(num_iters):
        model.fit(X,Y)
        if(i%20==0 or i==98):
            var = model.var_K
            Xs, mean, std = model.get_mean_std()
            plot(var, i+1, Xs, mean, std, X[:,0],Y[:,0], dir_name)
        x_new = model.optimize_acq_func(np.amax(Y))
        y_new = f(x_new)
        # y_new = model.predict(x_new)
        X = np.append(X, np.array(x_new).reshape(-1,1), axis=0)
        Y = np.append(Y, np.array(y_new).reshape(-1,1), axis=0)
        print(x_new, y_new)
    id_max = np.argmax(Y)
    return X[id_max], Y[id_max]

def plot(var, t, Xs, mean, std, Xsamples, Ysamples, dir_name):
    fig = plt.figure()
    plt.plot(Xs, mean, label=r'$\mu_{t}(x)$', color='orangered')
    plt.fill_between(Xs, mean-std, mean+std, facecolor='peachpuff', alpha=0.7)
    plt.scatter(Xsamples, Ysamples)
    plt.title(r'$\sigma^2 =$ {}, t = {}'.format(var, t))
    plt.legend()
    plt.savefig(os.path.join(dir_name,r'var_{}_t_{}.png'.format(var, t)))

    plt.close()

def main():

    # X = np.random.uniform(-5,5,100).reshape(-1,1)
    X = np.array([0]).reshape(-1,1)
    Y = np.array([f(x) for x in X]).reshape(-1,1)
    var_K = 30
    dir_name = os.path.join(os.getcwd(), "q52_var_{}".format(var_K))
    if(not os.path.exists(dir_name)):
        os.mkdir(dir_name)
    num_iters = 100
    gpr_model = GPR(var_K)
    x_max, f_max = bayes_optim(gpr_model, X, Y, num_iters, dir_name)
    print("({}, {})".format(x_max[0],f_max[0]))

main()
```

```
import numpy as np
from matplotlib import pyplot as plt
from q5_1 import *
import ipdb
import math
def calc_prob(x, lambda_, func):
    return np.exp(lambda_*func(x))

def MH_algo(lambda_, x, func, var):
    X = []
    x_ = x*1.0
    for i in range(500):
        while(1):
            y = np.random.normal(x_, var**0.5, 1)[0]
            if(y>=-5 and y<=5):
                break
        alpha = np.amin([calc_prob(y, lambda_, func)/calc_prob(x_, lambda_, func),1])
        if(math.isnan(alpha)):
            ipdb.set_trace()
        x_new = np.random.choice([y,x_], p=[alpha, 1-alpha])
        X.append(x_new*1.0)

    F = func(np.array(X))

    return X[np.argmax(F)], np.mean(X), np.mean(F)


def SA_algo(func, lambda_0, eta, var):

    x0 = np.random.uniform(-5, 5, 1)[0]
    x = x0
    lambda_ = lambda_0
    X = []
    F = []
    while (lambda_<=100):
        lambda_ = (1+eta)*lambda_
        x_new, x_mean, F_mean = MH_algo(lambda_, x, func, var)
        x = x_new*1.0
        X.append(x_mean*1.0)
        F.append(F_mean*1.0)

    return x, X, F

def plot_prop(data, prop_name):
    fig = plt.figure(figsize=(16,9))
    plt.plot(data)
    plt.xlabel("t")
    plt.ylabel(prop_name)
    plt.savefig("q5_3_{}.png".format(prop_name))
    plt.close()

def main():
    lambda_0 = 0.01
    eta = 0.1
    var = 0.1

    x_optim, X, F = SA_algo(f, lambda_0, eta, var)
    print(x_optim)
    plot_prop(X, "sampled_points_mean")
    plot_prop(F,"sampled_function_value_mean")

main()
```