Problem 1

This week the focus was on Reproducing Kernel Hilbert Spaces (RKHS) and how to use them in regression problems. An RKHS is a space of functions in which the sampling operator is continuous. Utilizing the Riesz representation theorem, which states that linear continuous functionals can be represented as the inner product of two elements of the Hilbert Space ($F(x) = \langle x,c \rangle$ where $x,c \in S$), there exists a function, $k_\tau$, where $F_\tau(f) = f(\tau) = \langle f,k_\tau \rangle$. Expanding the definition of $k_\tau$ a kernel is created which treats $\tau$ as another parameter, $k_\tau(t) = k(\tau, t)$. Then integrating $k(\tau, t) * f(\tau)$ over $\tau$ allows us to recover the original function, f. With this fact, estimates of the function f can be made using a kernel function. The kernel function itself can be constructed with an orthobasis on the Hilbert space.

Kernel Regression utilizes RKHS to recover/estimate functions from existing datapoints or "samples". The original infinite-dimensional least squares problem we had minimizing over f is now directly solvable by replacing $f(tm)$ with $\langle f, k_{tm} \rangle$. The relevance of kernel regression is that we can now map our data points to an infinite dimensional Hilbert space instead of working with some finite dimensional, giving us more representational power, without increasing our computational load, since at the end of the day a matrix of inner products is created. For the kernel regression case the dimension is data x data, while in the basis regression it is the dimension of our finite number of basis choosen.

2. A functional on a Hilbert space $\mathcal{F}: S \to \mathbb{R}$ is bounded if there exists a constant $C$ such that

$$\mathcal{F}(x) \leq C \|x\|_S, \text{ for all } x \in S$$

Argue directly that the sampling (or point evaluation) operator on $L_2([0,1])$

$$\mathcal{F}_\tau(f) = f(\tau)$$

is not bounded.

Let us choose the following class of functions from $L_2([0,1])$

$$f_n(t) = \begin{cases} 0 & \text{if } t < -\frac{1}{2n} \\ \sqrt{n} & \text{if } -\frac{1}{2n} \leq t \leq \frac{1}{2n} \\ 0 & \text{if } t > \frac{1}{2n} \end{cases}$$

$$\|f_n\| = \left(\int_{-\infty}^{\infty} |f(t)|^2 dt\right)^{1/2}$$

$$= \left(\int_{-\frac{1}{2n}}^{\frac{1}{2n}} |\sqrt{n}|^2 dt\right)^{1/2} = \left(\int_{-\frac{1}{2n}}^{\frac{1}{2n}} n\, dt\right)^{1/2} = \left(nt\Big|_{-\frac{1}{2n}}^{\frac{1}{2n}}\right)^{1/2} = \left(n\left(\tfrac{1}{2n} + \tfrac{1}{2n}\right)\right)^{1/2} = \left(n\left(\tfrac{1}{n}\right)\right)^{1/2} = 1.$$

Thus the norm of this class of functions remains constant at 1. However, the sampling operation on $-\frac{1}{2n} \leq t \leq \frac{1}{2n}$ increases as a function of $n(\sqrt{n})$ and is thus not bounded.

$$\mathcal{F}_\tau(f) = f(\tau) \leq C\|f_s\|$$

$f(\tau) \leq C(1)$ is not true for all $\tau$ thus the sampling functional is not bounded.

3. Let $C'([0,1])$ be the space of functions on $[0,1]$ that is differentiable on $(0,1)$.

(a) Let $\mathbb{D}_\tau$ be the functional that takes a $f \in C'([0,1])$ and returns the derivative at location $\tau$: $\mathbb{D}_\tau(f) = f'(\tau)$. Is $\mathbb{D}_\tau$ linear? Continuous?

$$\mathbb{D}_\tau[af+bg] = (af+bg)'(\tau) = (af)'(\tau) + (bg)'(\tau) = af'(\tau) + bg'(\tau)$$
$$= a\mathbb{D}_\tau(f) + b\mathbb{D}_\tau(g) \checkmark$$

$$\|f-g\|_S \le \delta \Rightarrow |\mathbb{D}_\tau(f) - \mathbb{D}_\tau(g)| \le \varepsilon$$

Let us show that $\mathbb{D}_\tau$ is not continuous by showing the above is not true with a counter example.

Let $f_n(t) = \frac{1}{n}\sin(2\pi n t)$ and $f_{n+1}(t) = \frac{1}{n+1}\sin(2\pi(n+1)t)$.

Then $\|f_{n+1} - f_n\| = \left(\int_0^1 |f_{n+1}(t) - f_n(t)|^2 dt\right)^{1/2}$

$$= \left(\int_0^1 (f_{n+1}(t) - f_n(t))^2 dt\right)^{1/2}$$
$$= \left(\int_0^1 (f_{n+1}(t))^2 + f_n(t)^2 - 2 f_{n+1}(t) f_n(t)\right) dt\Big)^{1/2}$$
$$= \Big(\int_0^1 \tfrac{1}{(n+1)^2}\sin^2(2\pi(n+1)t)\,dt + \int_0^1 \tfrac{1}{n^2}\sin^2(2\pi n t)\,dt$$
$$- \int_0^1 2\left(\tfrac{1}{n+1}\right)\left(\tfrac{1}{n}\right)\sin(2\pi(n+1)t)\sin(2\pi t)\,dt\Big)^{1/2}$$
$$= \Big(\tfrac{1}{(n+1)^2}\int_0^1 \tfrac{1}{2} - \tfrac{\cos(2\cdot 2\pi(n+1)t)}{2}\,dt + \tfrac{1}{n^2}\int_0^1 \tfrac{1}{2} - \tfrac{\cos(2\cdot 2\pi n t)}{2}\,dt$$
$$- \left(\tfrac{1}{n+1}\right)\left(\tfrac{1}{n}\right)\int_0^1 \cos(2\pi(n+1)t - 2\pi t) - \cos(2\pi(n+1)t + 2\pi t)\,dt\Big)^{1/2}$$
$$= \Big[\tfrac{1}{(n+1)^2}\left[\tfrac{1}{2}t + \tfrac{1}{2}(4\pi(n+1))\sin(4\pi(n+1)t)\right]_0^1$$
$$+ \tfrac{1}{n^2}\left[\tfrac{1}{2}t + \tfrac{1}{2}(4\pi n)\sin(4\pi n t)\right]_0^1$$
$$- \tfrac{1}{(n+1)n}\left[-2\pi n t\sin(2\pi n t) + (2\pi(n+2)t\sin(2\pi(n+2)t)\right]_0^1\Big]^{1/2}$$
$$= \left(\frac{1}{2(n+1)^2} + \frac{1}{2n^2}\right)^{1/2}$$
$$\le \delta \text{ for some arbitrary } n$$

In fact as $n \to \infty$ then $\delta$ can be taken closer to zero.

In otherwords $\delta \to 0$ as $n \to \infty$.

$$|D_\tau(f) - D_\tau(g)| = |f'_{n+1}(\tau) - f'_n(\tau)|$$

$$= \left|\frac{1}{n+1}(-2\pi(n+1)\cos(2\pi(n+1)\tau) - \frac{1}{n}(2\pi n)\cos(2\pi n\tau)\right|$$

$$= |2\pi\cos(2\pi(n+1)\tau) - 2\pi\cos(2\pi n\tau)|$$

$$= \left|2\pi \cdot 2\sin\left(\frac{2\pi(2n+1)\tau}{2}\right)\cdot\sin\left(\frac{2\pi\tau}{2}\right)\right|$$

$$= |4\pi\sin(\pi(2n+1)\tau)\cdot\sin(\pi\tau)| = \gamma(n) \quad 0 \le \gamma \le 4\pi \ \forall\, n$$

As $n$ approaches infinity, $\|f_{n+1}(t) - f_n(t)\|$ will be bounded by a smaller and smaller $\delta$, but $|D_\tau(f_{n+1}) - D_\tau(f_n)|$ will oscillate between values of $0$ and $4\pi$ and not approach $0$ as $n \to \infty$. Thus $D_\tau$ is not continuous

(b) Let $\ell_\tau$ be the functional that takes a $f \in C^1([0,1])$ and returns the definite integral

$$\ell_\tau(f) = \int_0^\tau f(t)\,dt$$

Is $\ell_\tau$ linear? Continuous?

$$\ell_\tau(af+bg) = \int_0^\tau (af+bg)(t)\,dt = \int_0^\tau af(t)+bg(t)\,dt$$
$$= \int_0^\tau af(t)\,dt + \int_0^\tau bg(t)\,dt = a\ell_\tau(f) + b\ell_\tau(g) \checkmark$$

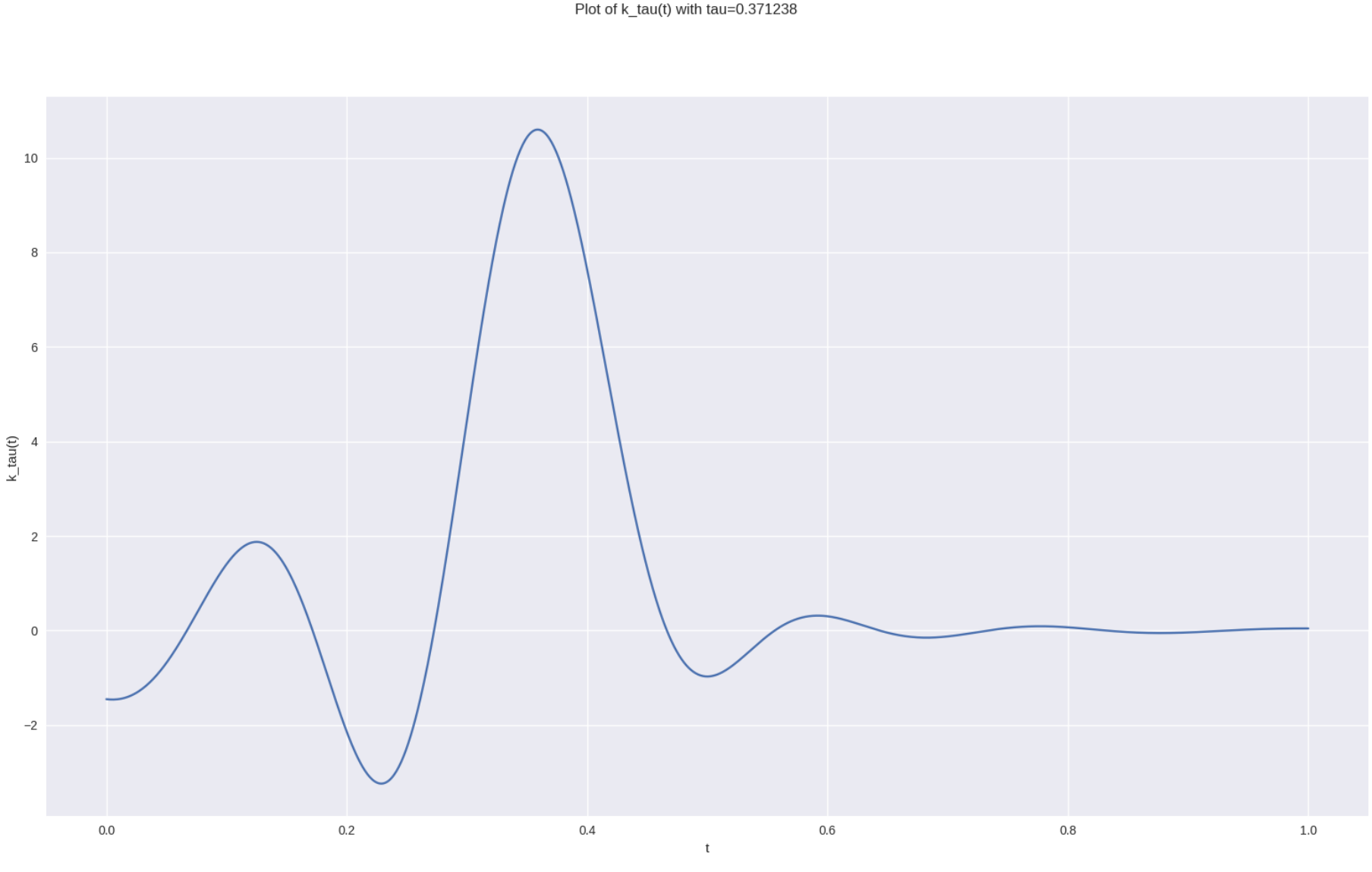
$$\|f-g\| \le \delta \Rightarrow |\ell_\tau(f)-\ell_\tau(g)| \le \varepsilon$$

$$|\ell_\tau(f)-\ell_\tau(g)| = \left|\int_0^\tau f(t)\,dt - \int_0^\tau g(t)\,dt\right|$$
$$= \left|\int_0^\tau f(t)-g(t)\,dt\right|$$
$$= \left|\int_0^\tau (f(t)-g(t))(1)\,dt\right|$$
$$= |\langle f-g, 1\rangle_\tau| \qquad \langle f,g\rangle_\tau = \int_0^\tau f(t)g(t)\,dt$$
$$= \left(|\langle f-g,1\rangle_\tau|^2\right)^{1/2}$$
$$\le \left(\langle f-g, f-g\rangle_\tau \cdot \langle 1,1\rangle_\tau\right)^{1/2} \text{ (cauchy-schwarz}$$
$$= \left(\int_0^\tau |f(t)-g(t)|^2\,dt \int_0^\tau |1|^2\,dt\right)^{1/2}$$
$$\le \left(\int_0^1 |f(t)-g(t)|^2\,dt \int_0^1 |1|^2\,dt\right)^{1/2} \text{ since } \tau \le 1$$
$$= \left(\|f-g\|\right)^{1/2}$$
$$\le \delta^{1/2}$$

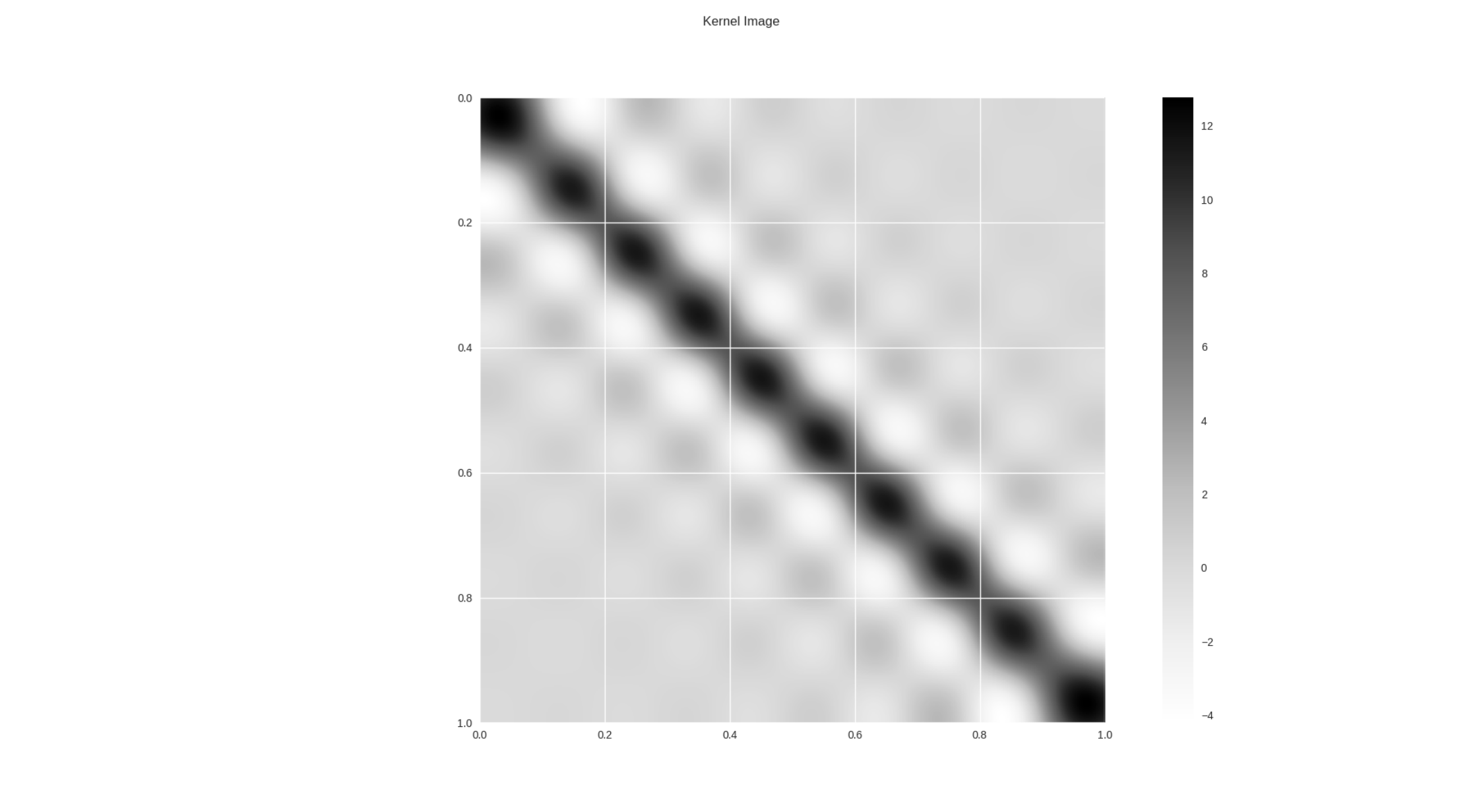
Letting $\varepsilon = \delta^{1/2}$ we can say if $\|f-g\| \le$ some $\delta$ then there exists an $\varepsilon$ s.t. $|\ell_\tau(f)-\ell_\tau(g)| \le \varepsilon$. Thus $\ell_\tau(\cdot)$ is continuous. $\checkmark$

```
import sys
import numpy as np
import matplotlib as mpl
import matplotlib.pyplot as plt
import scipy.io as sio
import scipy.integrate as integrate

mpl.style.use('seaborn')

phi = lambda t: np.exp(-t**2)
phi_n = lambda N,n,t: phi(N * t - (n + 1) + 0.5)
phi_n_tilde = lambda H,N,k,t: sum([H[k, n] * phi_n(N,n,t) for n in range(N)])
k = lambda H,N,s,t: \
    sum([phi_n(N,n,s) * phi_n_tilde(H,N,n,t) for n in range(N)])
f = lambda t,alpha_list: \
    sum([alpha_list[idx] * phi_n(len(alpha_list), idx, t) \
        for idx in range(len(alpha_list))])

def compute_H(N):
    G = np.ones(shape=(N,N))

    phii_phij = lambda t: phi_n(N, i, t) * phi_n(N, j, t)

    for i in range(N):
        for j in range(N):
            G[i, j] = integrate.quad(phii_phij, 0, 1)[0]

    H = np.linalg.inv(G)

    return H

def part_a():
    print("Part a")
    tau = 0.371238
    N = 10

    fig = plt.figure()
    fig.suptitle("Plot of k_tau(t) with tau=" + str(tau))
    ax = fig.add_subplot(111)

    t_vec = np.linspace(0, 1, 1000)

    H = compute_H(N)

    ax.plot(t_vec, k(H, N, tau, t_vec))

    ax.set_xlabel("t")
    ax.set_ylabel("k_tau(t)")

    plt.show()

    alpha_list = np.random.randn(N)

    f_k_tau = lambda t: f(t, alpha_list) * k(H, N, tau, t)
    f_inner_prod_k_tau = integrate.quad(f_k_tau, 0, 1)[0]
    f_tau = f(tau, alpha_list)

    print("<f,k_tau> = " + str(f_inner_prod_k_tau))
    print("f(tau) = " + str(f_tau))

part_a()

def part_b():
    length = 1000
```

```
    s_vec = np.linspace(0,1,length)
    t_vec = np.linspace(0,1,length)

    N = 10
    H = compute_H(N)

    K = np.ones(shape=(length,length))

    for i in range(length):
        for j in range(length):
            K[i, j] = k(H, N, s_vec[i], t_vec[j])

    fig = plt.figure()
    fig.suptitle("Kernel Image")
    ax = fig.add_subplot(111)

    kernel_image = ax.imshow(K, origin = 'upper', extent = [0, 1, 1, 0])
    fig.colorbar(kernel_image, ax=ax)
    plt.show()

part_b()
```


Plot of k_tau(t) with tau=0.371238
k_tau(t)
10
8
6
4
2
0
−2
0.0
0.2
0.4
0.6
0.8
1.0
t

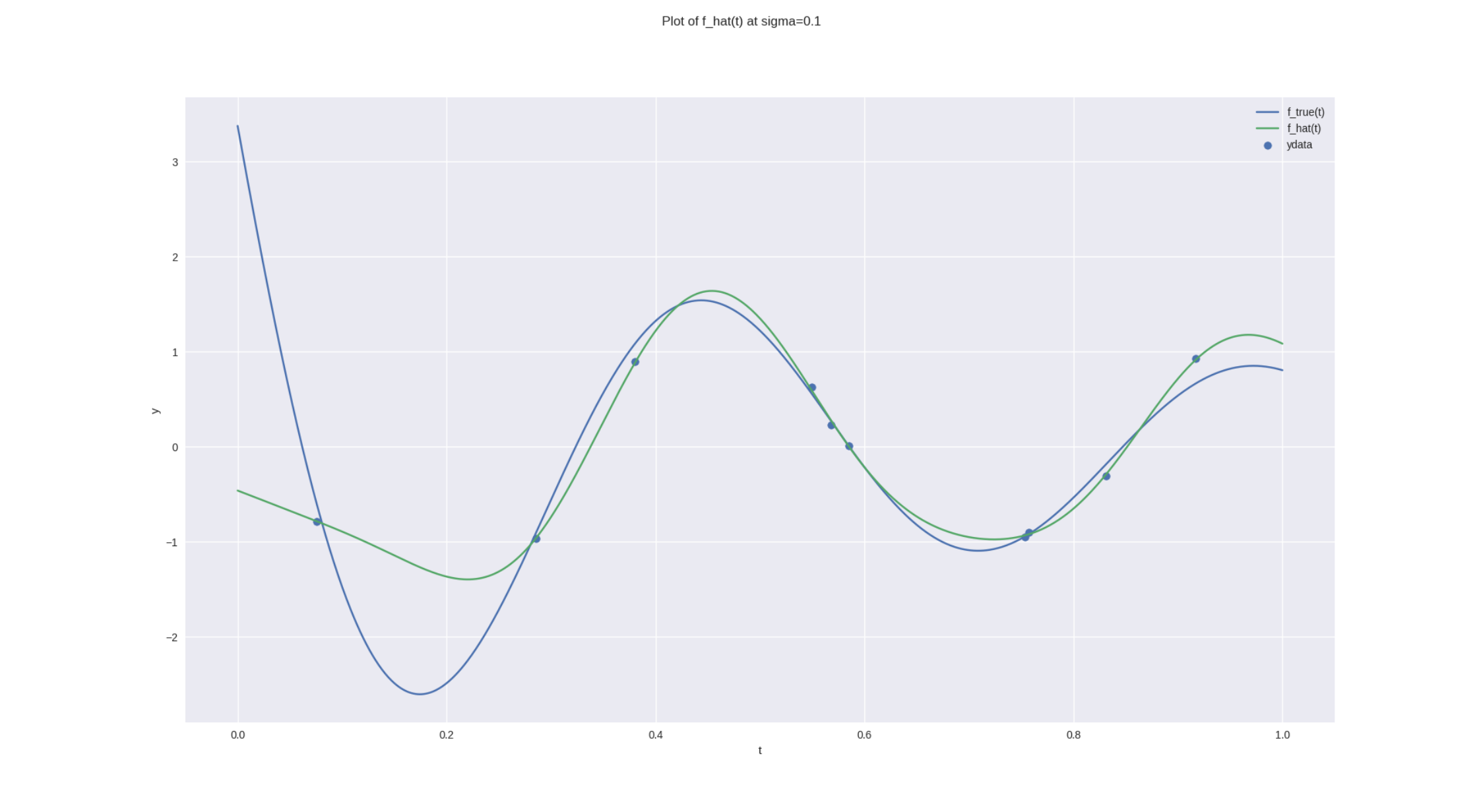
```
Part a
<f,k_tau> = -0.8127494521781716
f(tau) = -0.8127494521781734
```

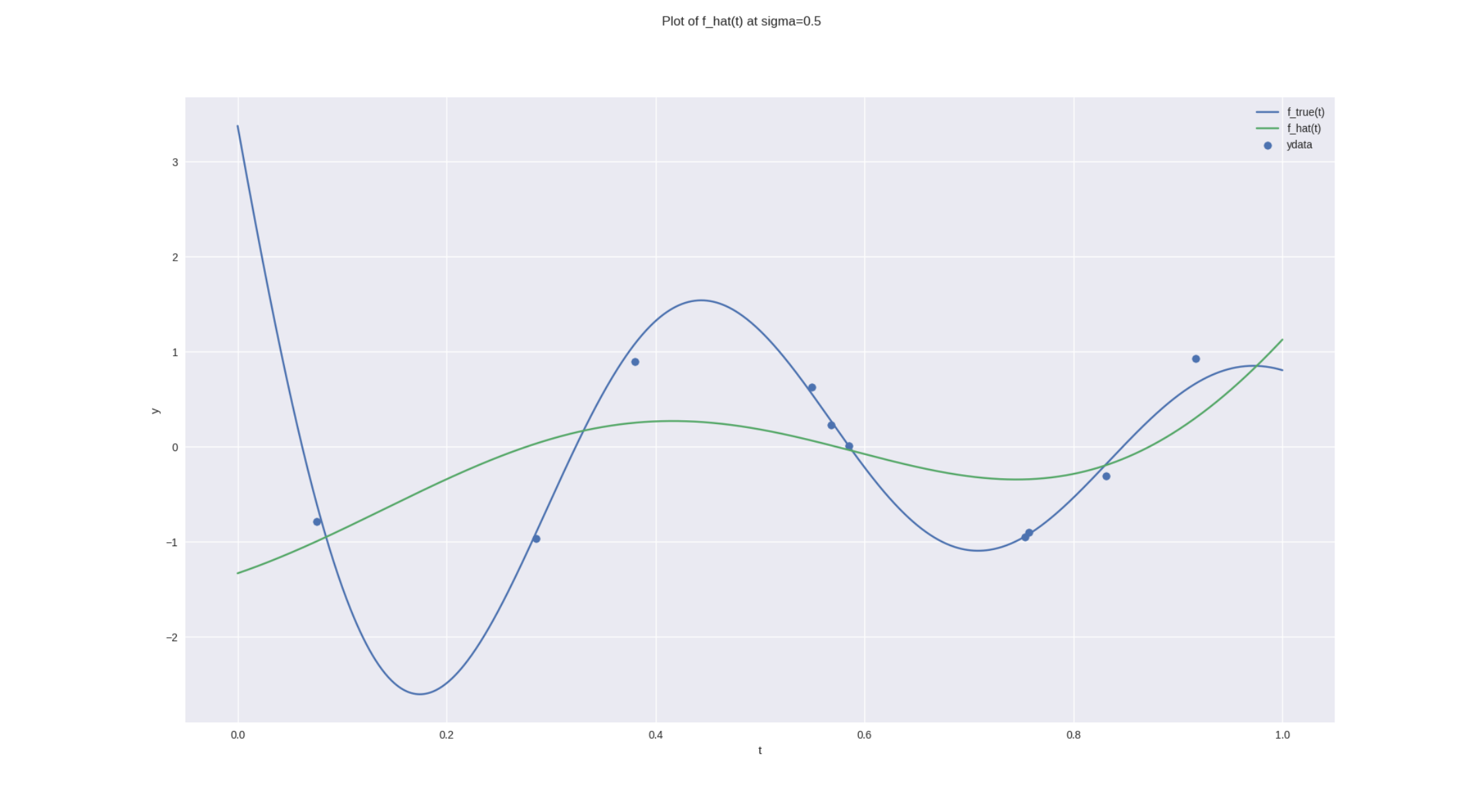

Kernel Image
0.0
0.2
0.4
0.6
0.8
1.0
0.0
0.2
0.4
0.6
0.8
1.0
12
10
8
6
4
2
0
−2
−4

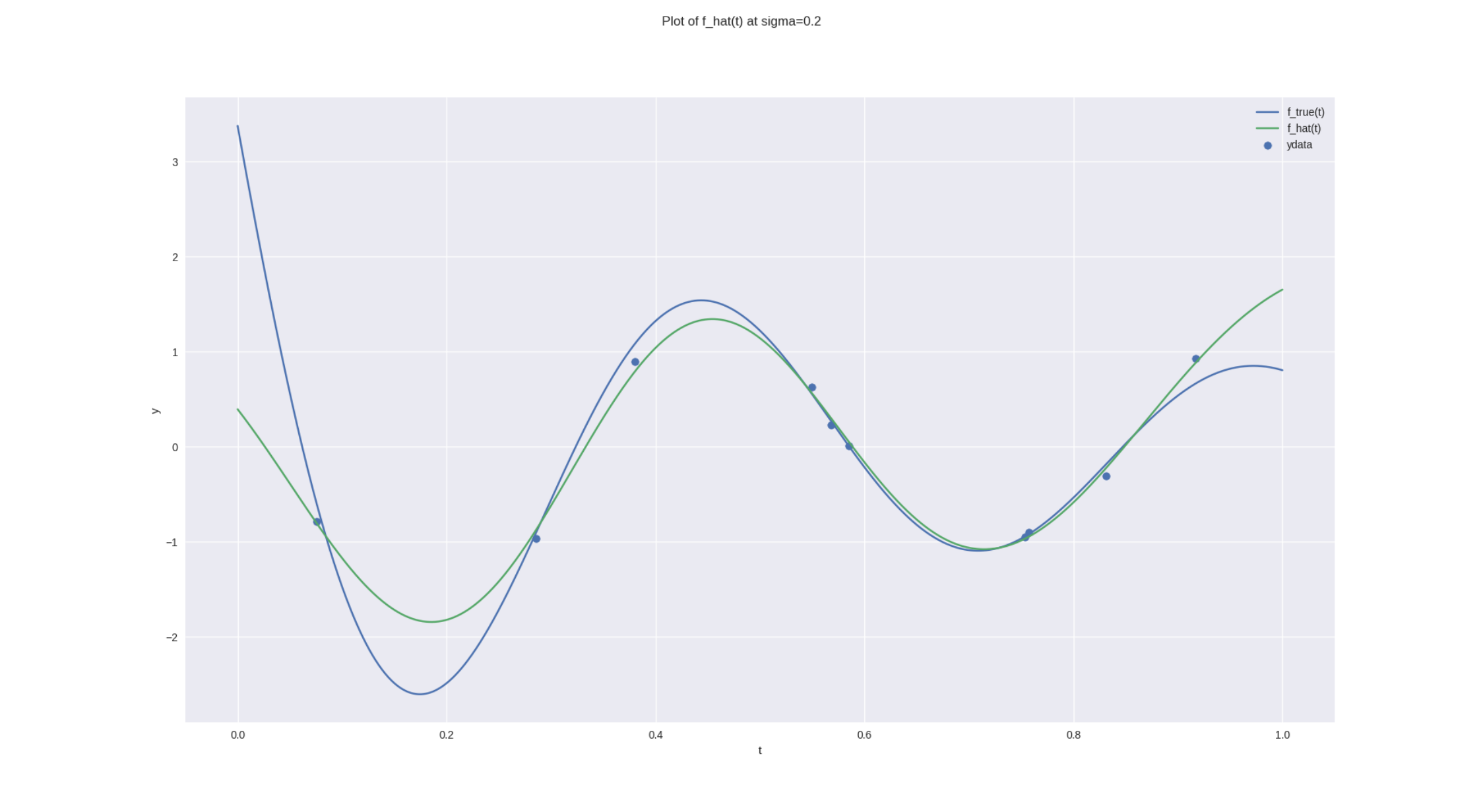
```
import sys
import numpy as np
import matplotlib as mpl
import matplotlib.pyplot as plt
import scipy.io as sio
import scipy.integrate as integrate

mpl.style.use('seaborn')

mat_filename ="hw05p5_data.mat"
data_samples = sio.loadmat(mat_filename)

tdata = data_samples['T']
ydata = data_samples['y']

f_true = lambda t: (np.sin(12 * (t + 0.2)))/(t + 0.2)
k = lambda s,t,sigma: np.exp(-1 * np.abs(t - s)**2 / (2 * sigma**2))
f_hat = lambda t,alpha_list,t_list,sigma: \
    sum([alpha_list[idx] * k(t_list[idx],t,sigma) for idx in
range(len(alpha_list))])

def compute_alpha_list(delta, tdata, ydata, sigma):
    M = len(ydata)

    K = np.ones(shape=(M,M))

    ki_kj = lambda t: k(t,tdata[i],sigma) * k(t,tdata[j],sigma)

    for i in range(0, M):
        for j in range(0, M):
            K[i,j] = k(tdata[i],tdata[j],sigma)

    alpha = np.linalg.inv(K + delta * np.identity(M)) @ ydata
    return alpha

def plot(alpha_list, sigma, tdata, ydata):
    fig = plt.figure()
    fig.suptitle("Plot of f_hat(t) at sigma=" + str(sigma))
    ax = fig.add_subplot(111)

    t_vec = np.linspace(0, 1, 1000)

    ax.plot(t_vec, f_true(t_vec), label="f_true(t)")
    ax.scatter(tdata, ydata, label="ydata")
    ax.plot(t_vec, [f_hat(t, alpha_list, tdata, sigma) for t in t_vec], \
            label="f_hat(t)")

    ax.set_xlabel("t")
    ax.set_ylabel("y")
    ax.legend()

    plt.show()

def sample_error(tdata, ydata, alpha_list, sigma):
    error_sum = 0
    for m in range(len(tdata)):
        error_sum += np.abs(ydata[m] - f_hat(tdata[m], alpha_list, tdata,
sigma))**2
    return np.sqrt(error_sum)

def generalization_error(alpha_list, tdata, sigma):
    f_error = lambda z: \
            (np.abs(f_hat(z, alpha_list, tdata, sigma) - f_true(z)))**2
    return np.sqrt(integrate.quad(f_error, 0, 1)[0])
```

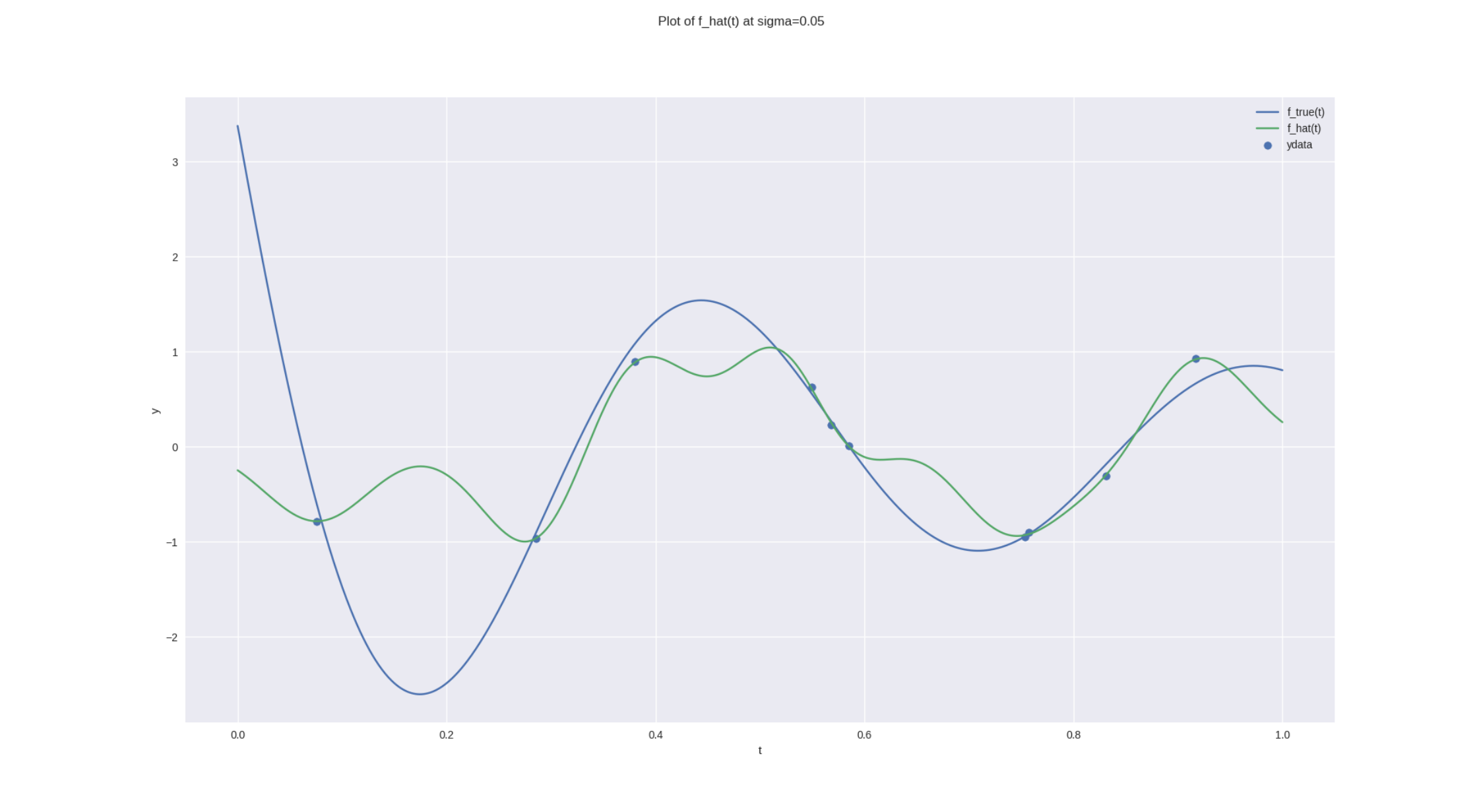
```
def part_a():
    print("Part (a)")
    sigma = 1/10
    delta = 0.004

    alpha_list = compute_alpha_list(delta, tdata, ydata, sigma)

    plot(alpha_list, sigma, tdata, ydata)

    sample_error_value = sample_error(tdata, ydata, alpha_list, sigma)
    generalization_error_value = generalization_error(alpha_list, tdata, sigma)

    print("sample error (sigma=" + str(sigma) + "): " \
            + str(sample_error_value))
    print("generalization error (sigma=" + str(sigma) + "): " \
        + str(generalization_error_value))

    print("""
A sigma value that is too low, will have a lower sample error but it will also
have a higher generalization error since it will overfit the data. This is
because small sigma values will result in a kernel function that is "tighter"
and can predict to a high degree, but will also mean it won't be able to
generalize the data points close to it. A sigma value that is too high on the
other hand will result in a "fatter" kernel function that can approximate values
close to the given data points but will not be as accurate for the given data
points. This particular sigma finds a good balance between the two.
""")

part_a()

def part_b(tdata_og,ydata_og):
    print("Part (b)")
    sigma_list = [1/2, 1/5, 1/20, 1/50, 1/100, 1/200]
    delta = 0.004

    for sigma in sigma_list:

        alpha_list = compute_alpha_list(delta, tdata, ydata, sigma)

        plot(alpha_list, sigma, tdata, ydata)

        sample_error_value = sample_error(tdata, ydata, alpha_list, sigma)
        generalization_error_value = generalization_error(alpha_list, tdata,
sigma)
        print("sample error (sigma=" + str(sigma) + "): " \
                + str(sample_error_value))
        print("generalization error (sigma=" + str(sigma) + "): " \
            + str(generalization_error_value))

    plt.show()

    print("""
If we have a small number of datapoints that it makes sense for us to choose a
larger value of sigma. This is because at large values of sigma, we have a
"fatter" kernel function and thus can generalize better than "tighter" kernel
functions. At a large number of datapoints, it makes sense to leverage the dense
dataset we have of the underlying function. This means we can decrease the sigma
value to obtain accurate predictions at the given datapoints while maintaining
a large number of kernel functions to approximate the underlying function. This
is true because our predicted function is a linear combination of the kernel
function at each datapoint.
Note: Here I am referring to kernel functions/kernel as the bump basis/Gaussian
distribution for this problem.
```

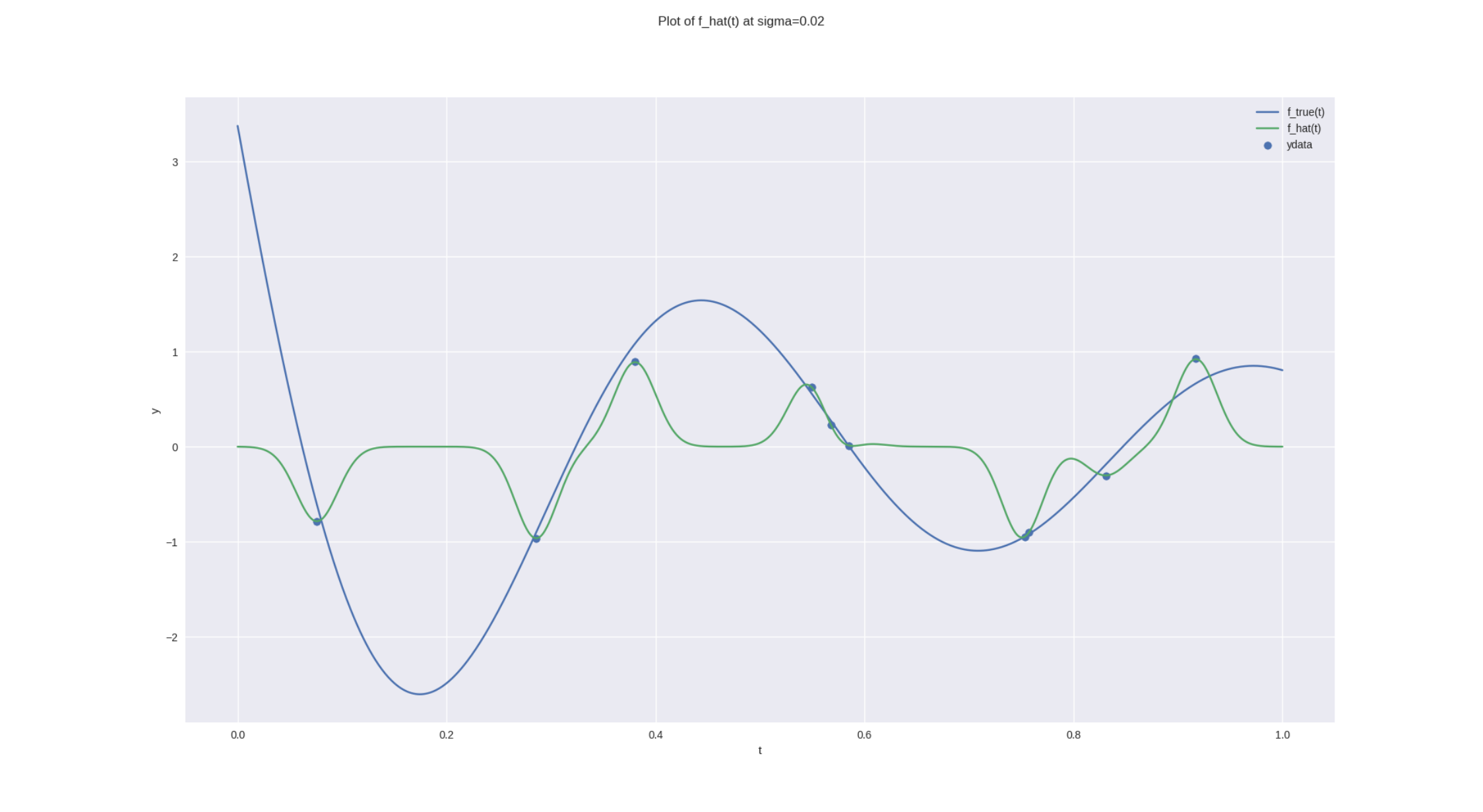
```
""")

part_b(tdata,ydata)
```

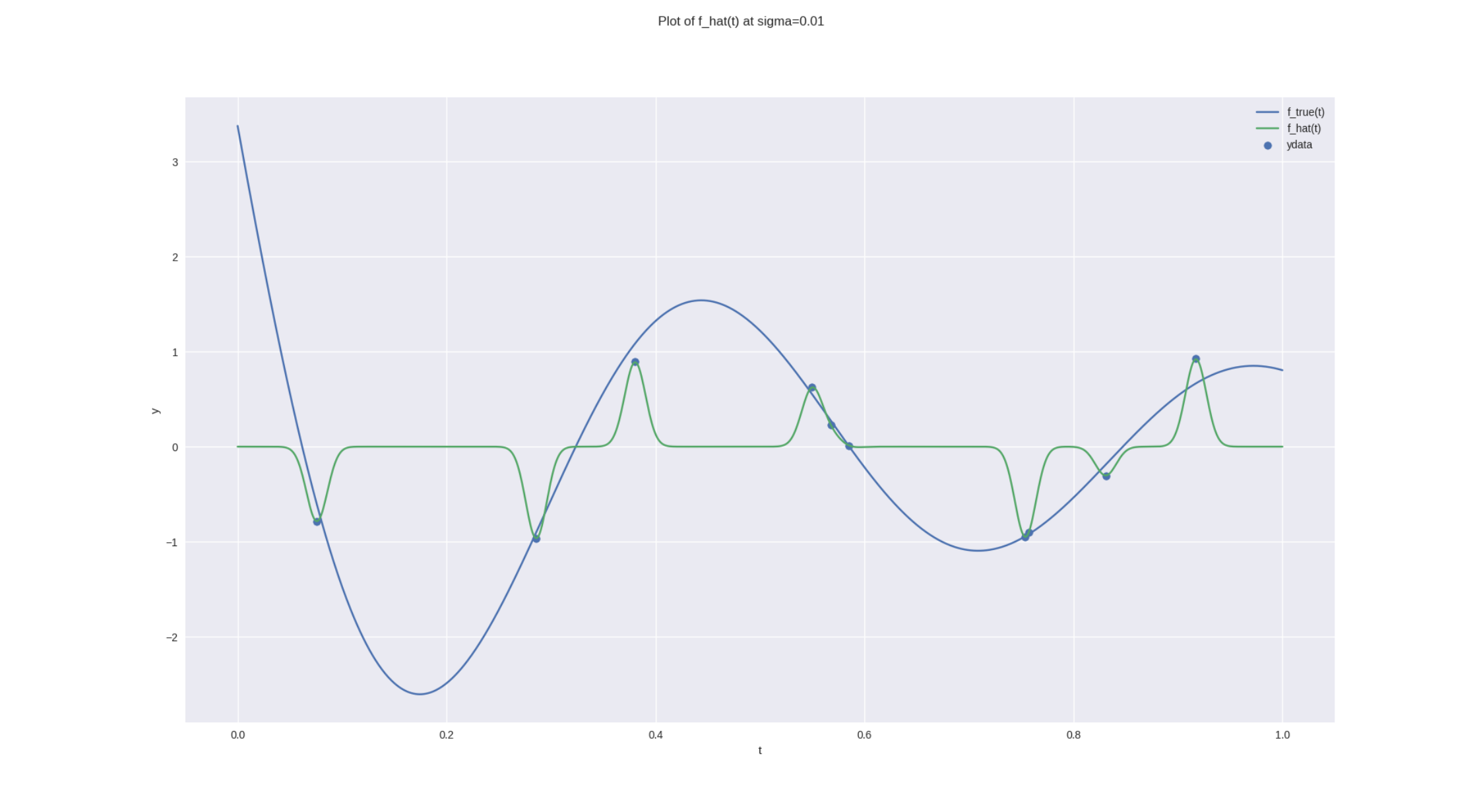

Plot of f_hat(t) at sigma=0.1
f_true(t)
f_hat(t)
ydata
y
t
3
2
1
0
−1
−2
0.0
0.2
0.4
0.6
0.8
1.0

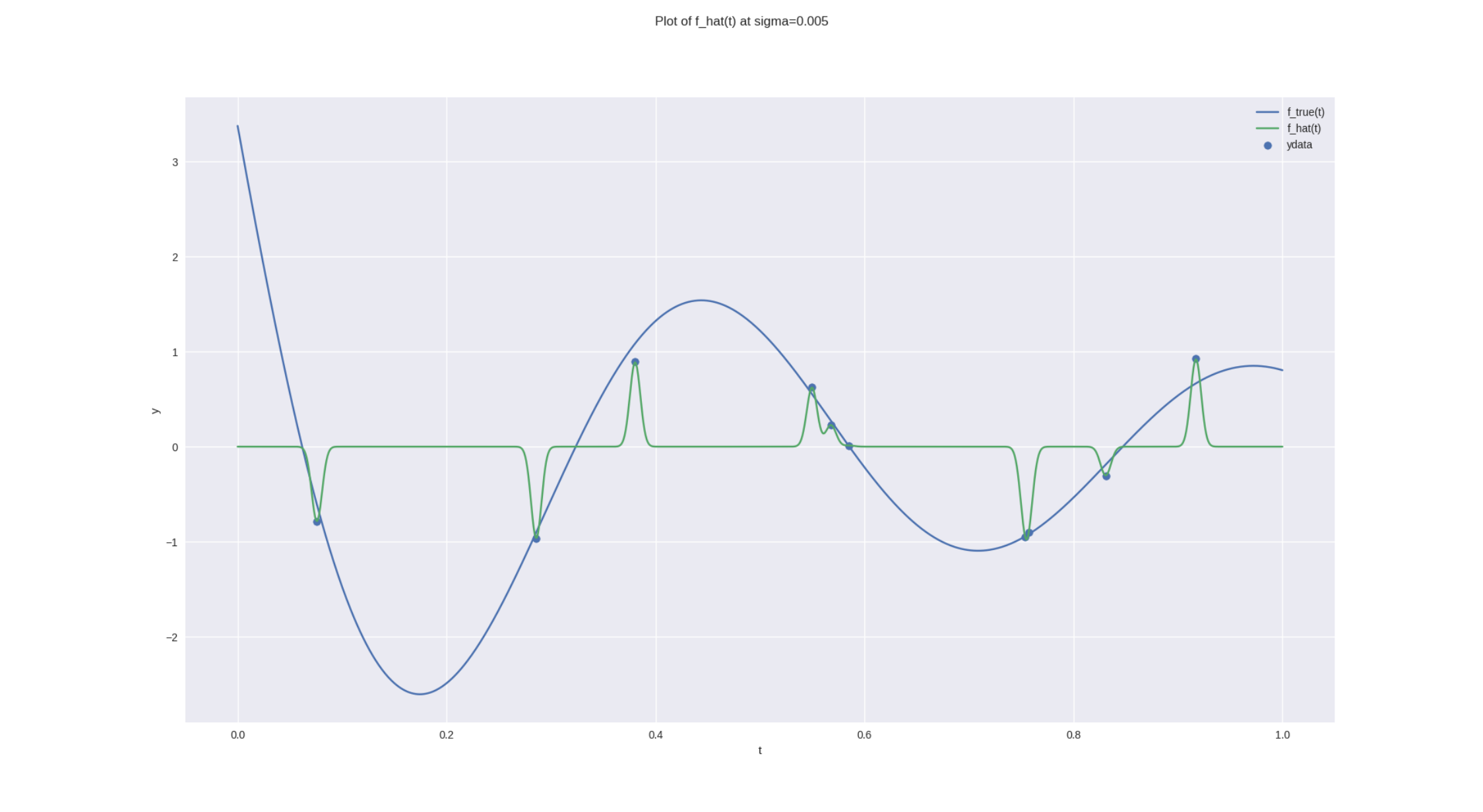

Plot of f_hat(t) at sigma=0.5
f_true(t)
f_hat(t)
ydata
y
t
3
2
1
0
−1
−2
0.0
0.2
0.4
0.6
0.8
1.0


Plot of f_hat(t) at sigma=0.2
f_true(t)
f_hat(t)
ydata
y
t
3
2
1
0
−1
−2
0.0
0.2
0.4
0.6
0.8
1.0


Plot of f_hat(t) at sigma=0.05
f_true(t)
f_hat(t)
ydata
y
t
3
2
1
0
−1
−2
0.0
0.2
0.4
0.6
0.8
1.0


Plot of f_hat(t) at sigma=0.02
f_true(t)
f_hat(t)
ydata
y
t
3
2
1
0
−1
−2
0.0
0.2
0.4
0.6
0.8
1.0


Plot of f_hat(t) at sigma=0.01
f_true(t)
f_hat(t)
ydata
y
t
3
2
1
0
−1
−2
0.0
0.2
0.4
0.6
0.8
1.0


Plot of f_hat(t) at sigma=0.005
f_true(t)
f_hat(t)
ydata
y
t
3
2
1
0
−1
−2
0.0
0.2
0.4
0.6
0.8
1.0

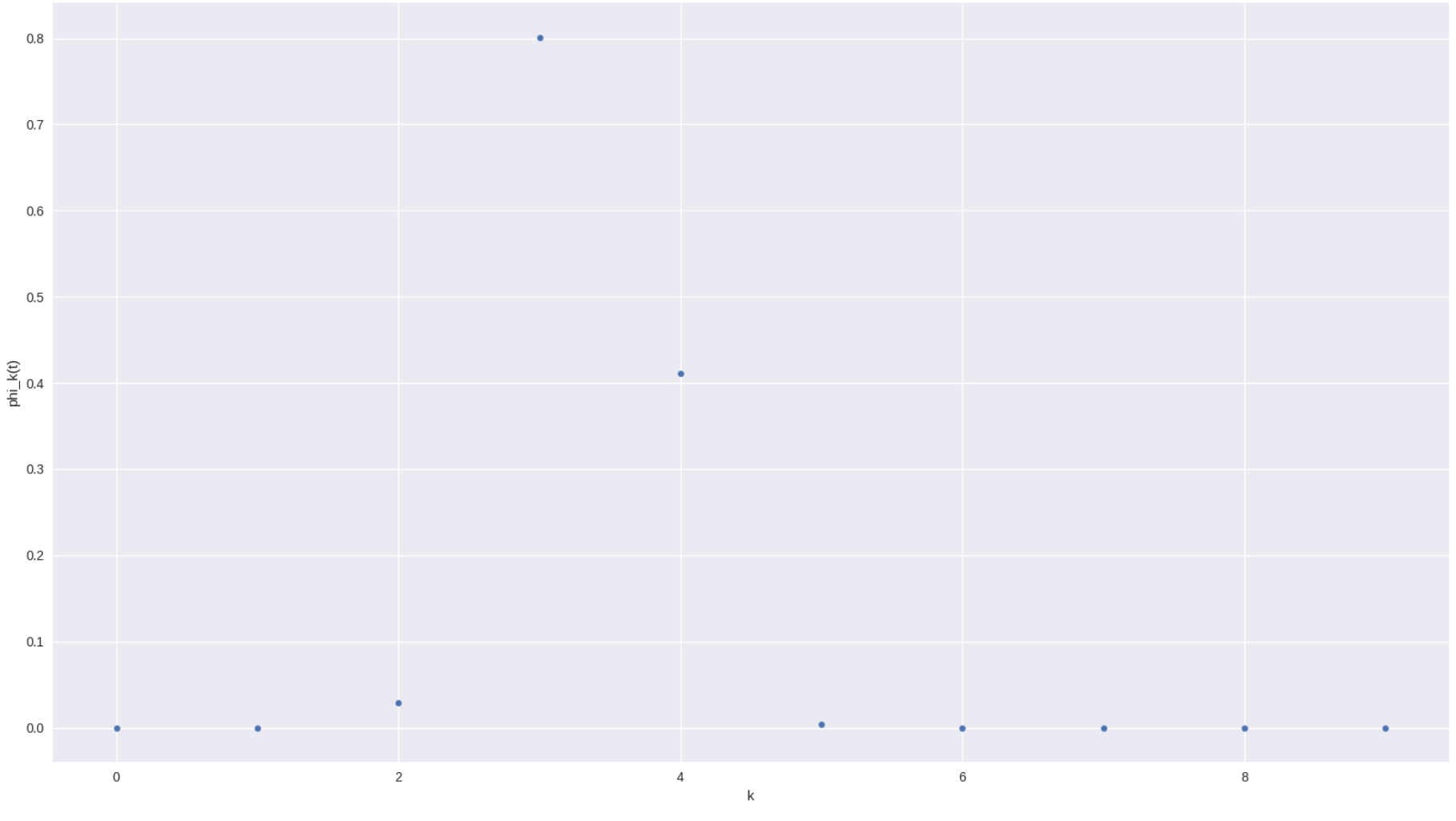
Part (a)
sample error (sigma=0.1): [0.07240677]
generalization error (sigma=0.1): 0.7431462241572463

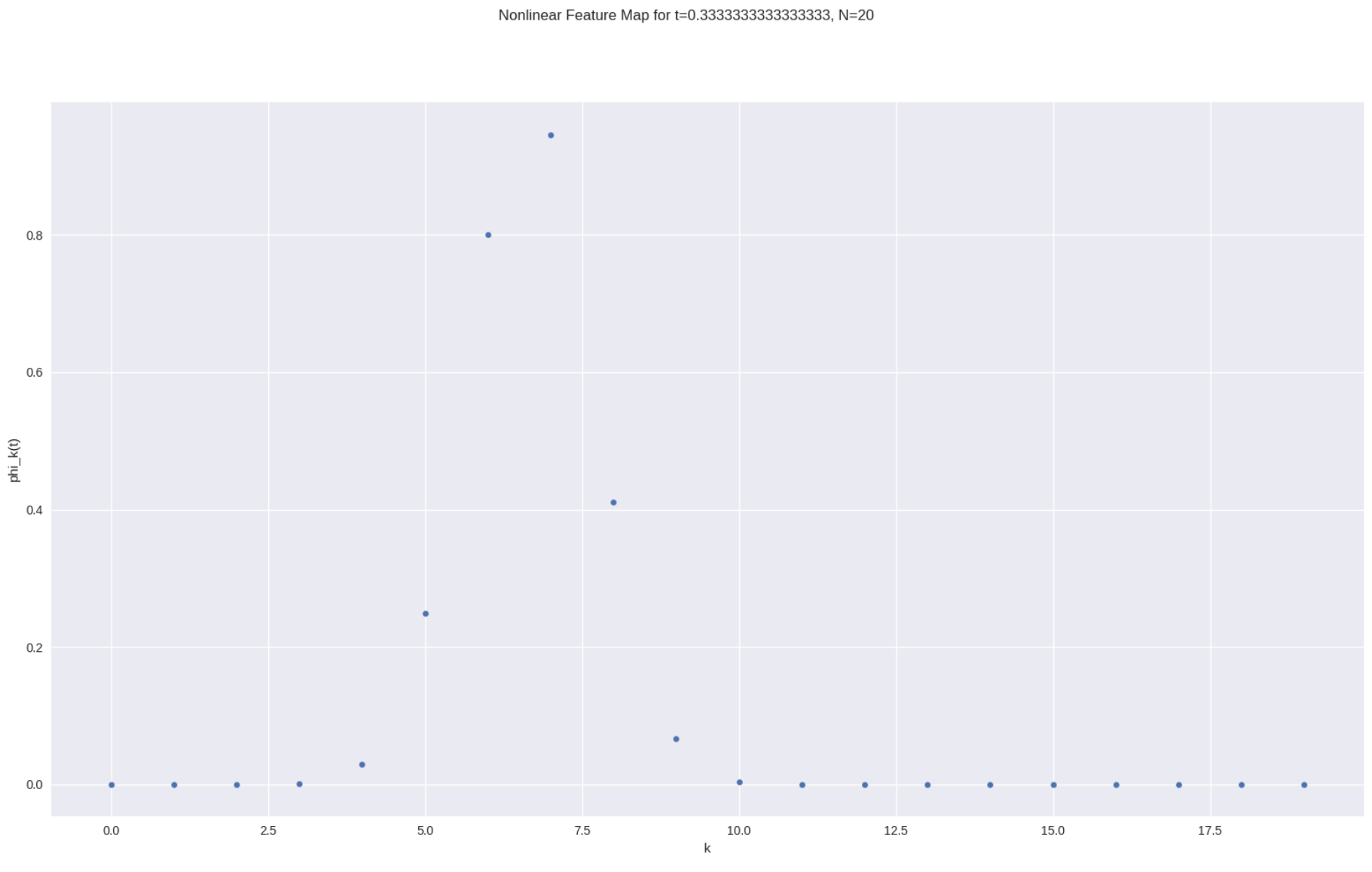
A sigma value that is too low, will have a lower sample error but it will also have a higher generalization error since it will overfit the data. This is because small sigma values will result in a kernel function that is "tighter" and can predict to a high degree, but will also mean it won't be able to generalize the data points close to it. A sigma value that is too high on the other hand will result in a "fatter" kernel function that can approximate values close to the given data points but will not be as accurate for the given data points. This particular sigma finds a good balance between the two.

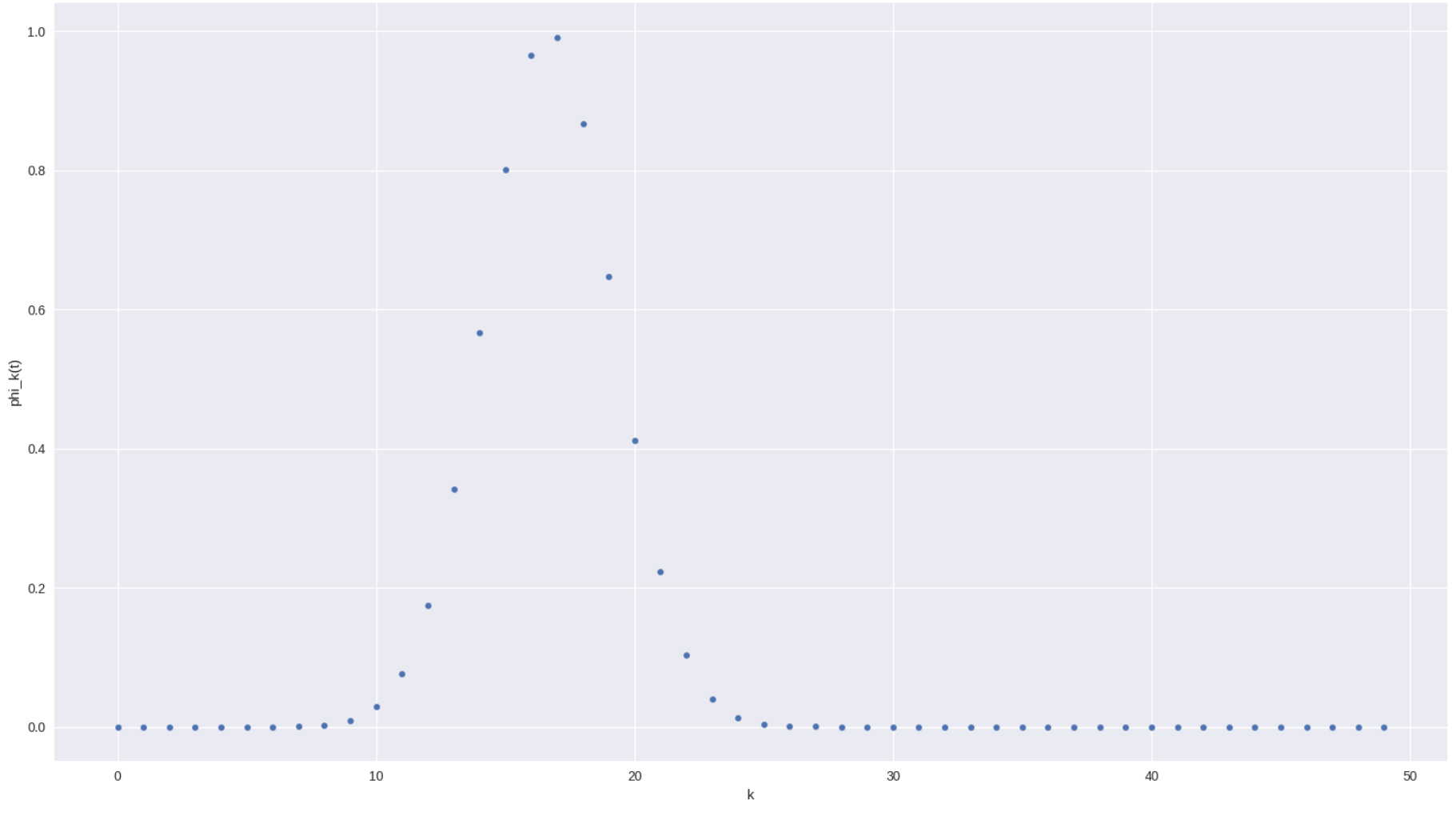
Part (b)
sample error (sigma=0.5): [1.70268885]
generalization error (sigma=0.5): 1.1997896243085355
sample error (sigma=0.2): [0.19779599]
generalization error (sigma=0.2): 0.5533227772504742
sample error (sigma=0.05): [0.04785688]
generalization error (sigma=0.05): 0.9834890574524433
sample error (sigma=0.02): [0.01151915]
generalization error (sigma=0.02): 1.178979244967919
sample error (sigma=0.01): [0.00841719]
generalization error (sigma=0.01): 1.2294657654316712
sample error (sigma=0.005): [0.00827552]
generalization error (sigma=0.005): 1.2545027068980132

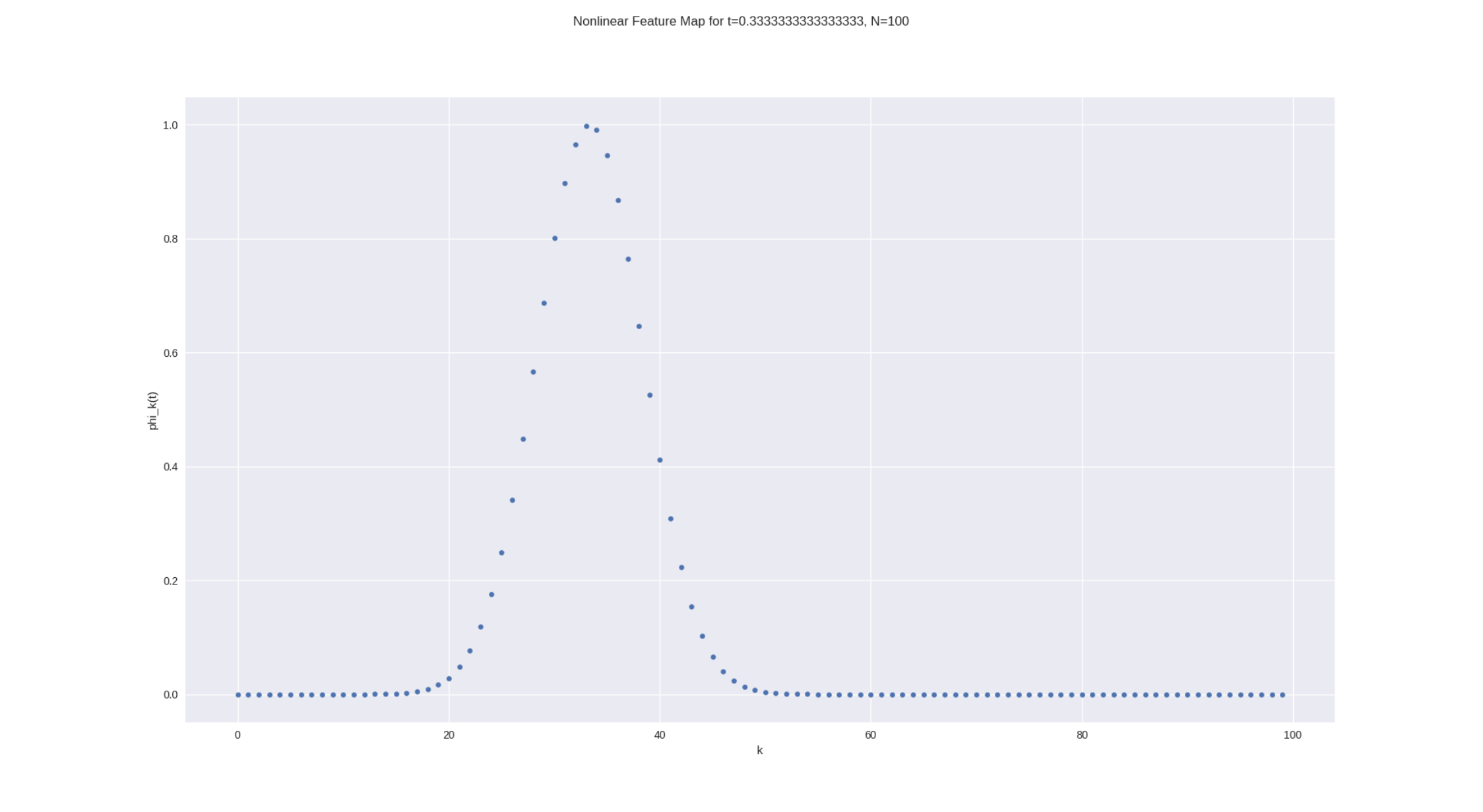
If we have a small number of datapoints that it makes sense for us to choose a larger value of sigma. This is because at large values of sigma, we have a "fatter" kernel function and thus can generalize better than "tighter" kernel functions. At a large number of datapoints, it makes sense to leverage the dense dataset we have of the underlying function. This means we can decrease the sigma value to obtain accurate predictions at the given datapoints while maintaining a large number of kernel functions to approximate the underlying function. This is true because our predicted function is a linear combination of the kernel function at each datapoint.
Note: Here I am referring to kernel functions/kernel as the bump basis/Gaussian distribution for this problem.

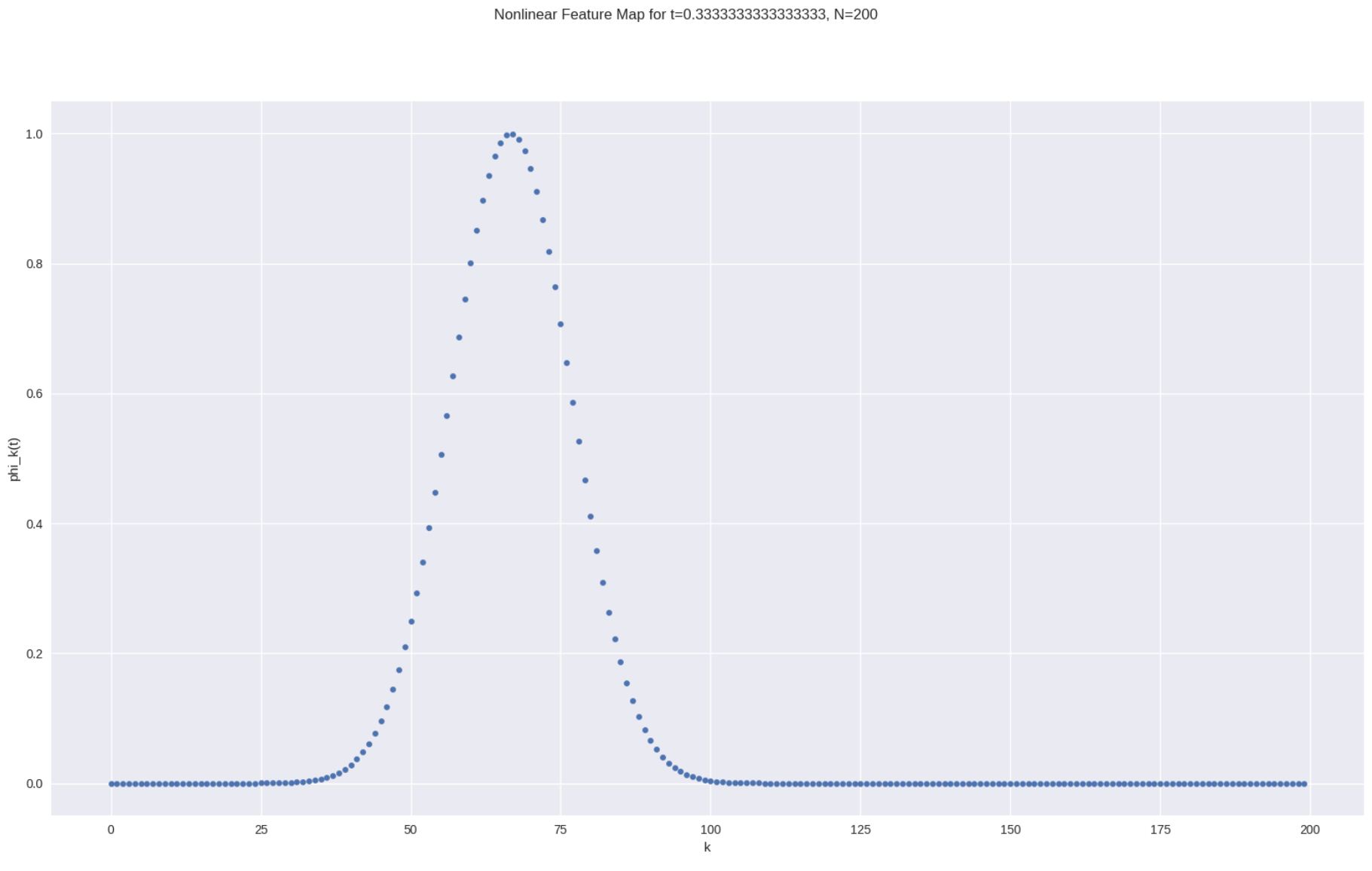
```
import sys
import numpy as np
import matplotlib as mpl
import matplotlib.pyplot as plt

mpl.style.use('seaborn')

def prob6():
    t = 1/3
    N_list = [10, 20, 50, 100, 200]

    for N in N_list:
        g = lambda z: np.exp(-200*z**2)
        phi_k = lambda z,k: g(z - k/N)
        Psi = lambda z: [phi_k(z,k) for k in range(N)]

        fig = plt.figure()
        fig.suptitle("Nonlinear Feature Map for t=" + str(t) + ", N=" + str(N))
        ax = fig.add_subplot(111)
        ax.scatter(range(N), Psi(t), s = 20)

        ax.set_xlabel("k")
        ax.set_ylabel("phi_k(t)")

        plt.show()

    k = lambda z,y: np.exp(-100 * np.abs(z - y)**2)
    Phi_t = lambda z: k(z,t)

    fig = plt.figure()
    fig.suptitle("Radial Basis Kernel Map for t=" + str(t))
    ax = fig.add_subplot(111)
    t_vec = np.linspace(0,1,1000)
    ax.plot(t_vec, Phi_t(t_vec))

    ax.set_xlabel("s")
    ax.set_ylabel("Phi_t(t)")

    plt.show()

    print("""
Here we see that with nonlinear regression using a basis our feature map gives
us a discrete set of basis functions to work with. However, with the kernel
regression with a Gaussian radial basis function we are able to create a
continuous feature map. One way to think about this is that the feature map of
the nonlinear function gives us a finite set of basis functions to aid in
prediction, while for the kernel regression the feature map is continuous and
provides an infinite number of basis functions to use for prediction. As N
increases, the nonliner feature map approaches the profile of the radial basis
kernel map.
""")

prob6()
```

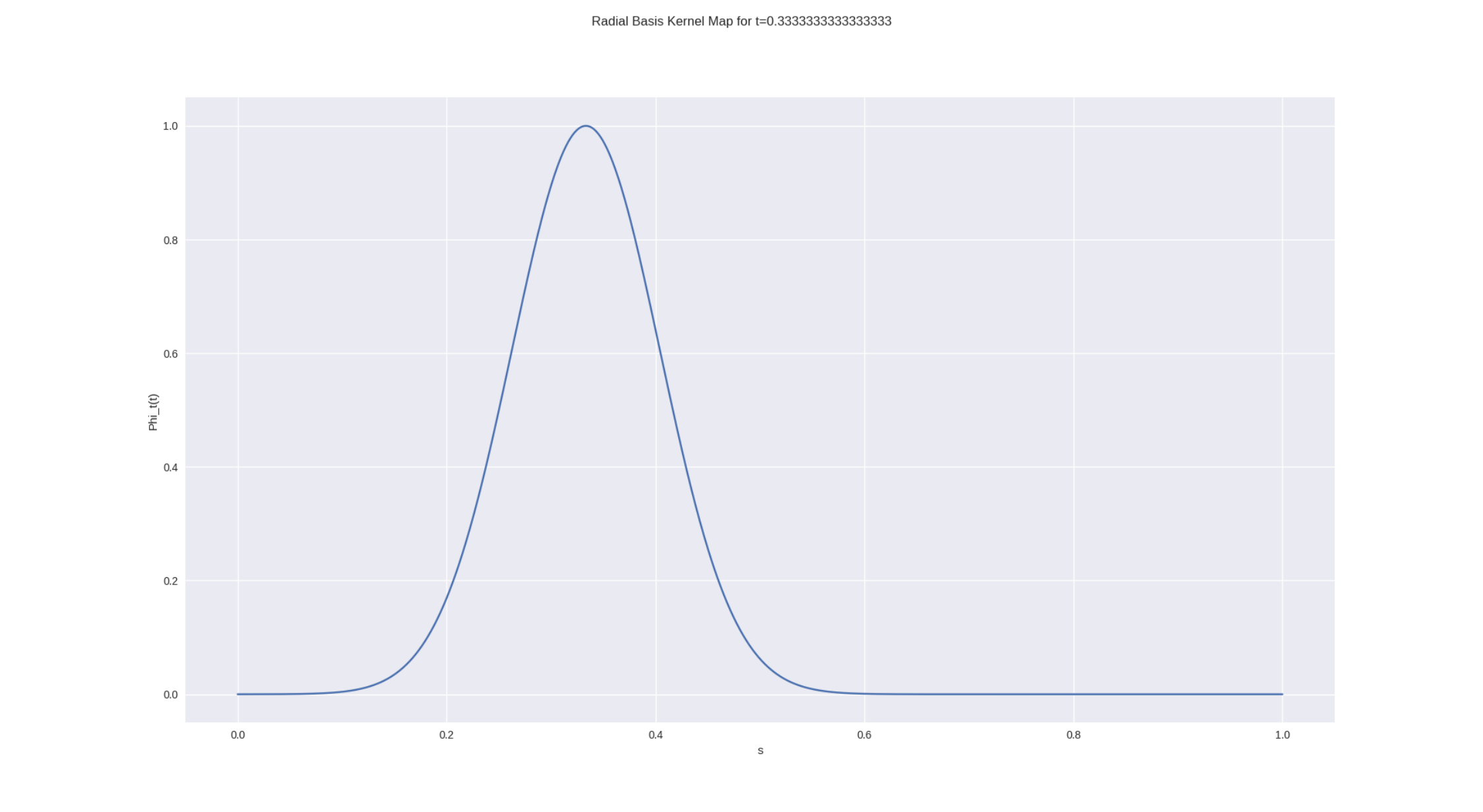

Nonlinear Feature Map for t=0.3333333333333333, N=10
phi_k(t)
0.0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0
2
4
6
8
k


Nonlinear Feature Map for t=0.3333333333333333, N=20
phi_k(t)
0.0
0.2
0.4
0.6
0.8
k
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5


Nonlinear Feature Map for t=0.3333333333333333, N=50
phi_k(t)
k
0.0
0.2
0.4
0.6
0.8
1.0
0
10
20
30
40
50


Nonlinear Feature Map for t=0.3333333333333333, N=100
phi_k(t)
k
0.0
0.2
0.4
0.6
0.8
1.0
0
20
40
60
80
100


Nonlinear Feature Map for t=0.3333333333333333, N=200
phi_k(t)
k
0.0
0.2
0.4
0.6
0.8
1.0
0
25
50
75
100
125
150
175
200


Radial Basis Kernel Map for t=0.3333333333333333
1.0
0.8
0.6
0.4
0.2
0.0
Phi_t(t)
0.0
0.2
0.4
0.6
0.8
1.0
s

Here we see that with nonlinear regression using a basis our feature map gives us a discrete set of basis functions to work with. However, with the kernel regression with a Gaussian radial basis function we are able to create a continuous feature map. One way to think about this is that the feature map of the nonlinear function gives us a finite set of basis functions to aid in prediction, while for the kernel regression the feature map is continuous and provides an infinite number of basis functions to use for prediction. As N increases, the nonliner feature map approaches the profile of the radial basis kernel map.