Table of Contents

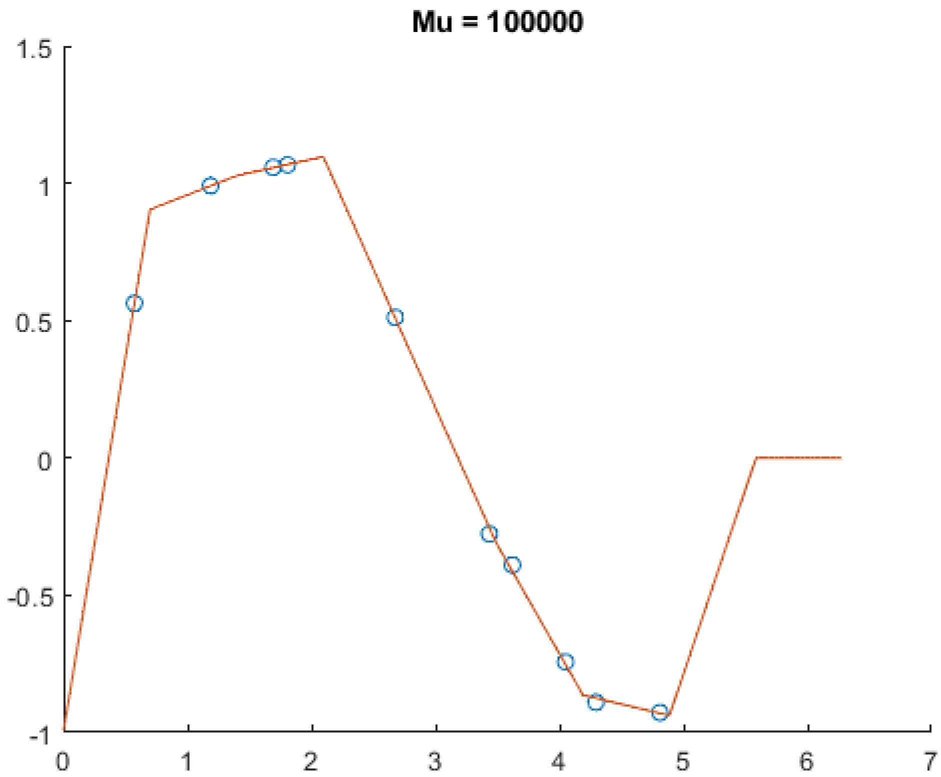

```
clear; clc;

load("simple.mat")
```

Part 2

Train the model on the data in simple.mat using ten hat functions and $\mu = 105$. Plot and turn in the learned model (the function fit to the data) on the interval $[0,2\pi]$.
****

```
params = hat_basis(0, 2 * pi, 10);
[~, M] = size(params);
func = @func_hat;

mu = 10^(5);
w = lsefit(x, t, params, func, mu);

x_test = (0:0.01:2* pi)';  % Sample points to look at function
[N, ~] = size(x_test);
Sig_test =  eval_basis(params, func, x_test);

y = Sig_test * w;

figure();
hold on
scatter(x, t);
plot(x_test, y);
```

```
title("Mu = " + mu);
hold off
```

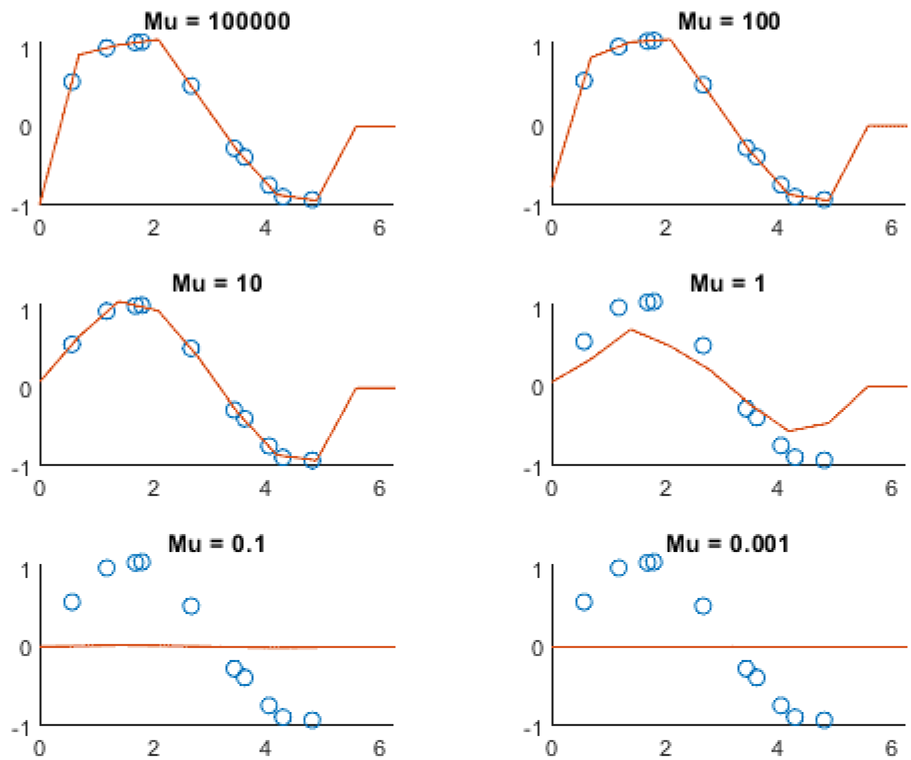


Part 3

Do the same for other values of the hyperparameter such as μ = 10 and μ = 1.
****
I notice that as mu becomes small the function gets a lot more regularized. So at high values of mu the function is overfit, while at low values of mu it is underfit and becomes a line.

```
mus = [10^5, 100, 10, 1, 0.1, 0.001];
figure()
tiledlayout(3,2)
for mu = mus
    params = hat_basis(0, 2 * pi, 10);
    [~, M] = size(params);
    func = @func_hat;
    w = lsefit(x, t, params, func, mu);

    x_test = (0:0.01:2* pi)';  % Sample points to look at function
    [N, ~] = size(x_test);
    Sig_test =  eval_basis(params, func, x_test);

    y = Sig_test * w;

    nexttile
```

```
    hold on
    scatter(x, t);
    plot(x_test, y);
    title("Mu = " + mu);
    hold off
end
```

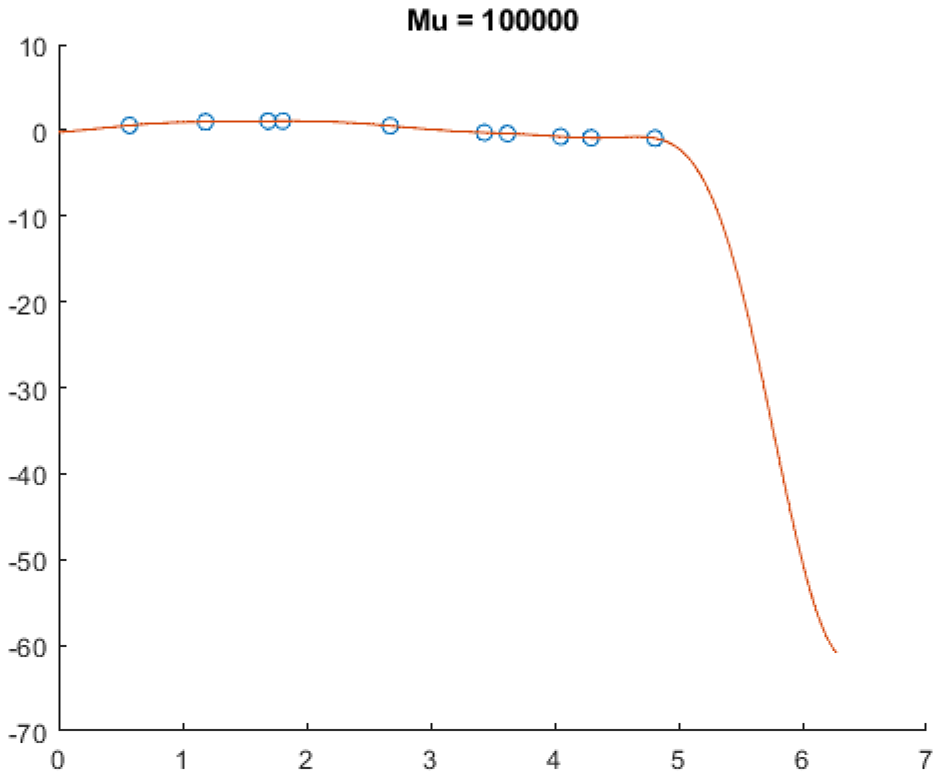


Part 5

Train the model on the data in simple.mat using ten gauss functions and $\mu = 105$. Plot and turn in the learned model (the function fit to the data) on the interval $[0,2\pi]$.
****

```
params = gauss_basis(0, 2 * pi, 10);
[~, M] = size(params);
func = @func_gauss;

mu = 10^(5);
w = lsefit(x, t, params, func, mu);

x_test = (0:0.01:2* pi)';  % Sample points to look at function
[N, ~] = size(x_test);
Sig_test =  eval_basis(params, func, x_test);

y = Sig_test * w;
```

```
figure();
hold on
scatter(x, t);
plot(x_test, y);
title("Mu = " + mu);
hold off
```



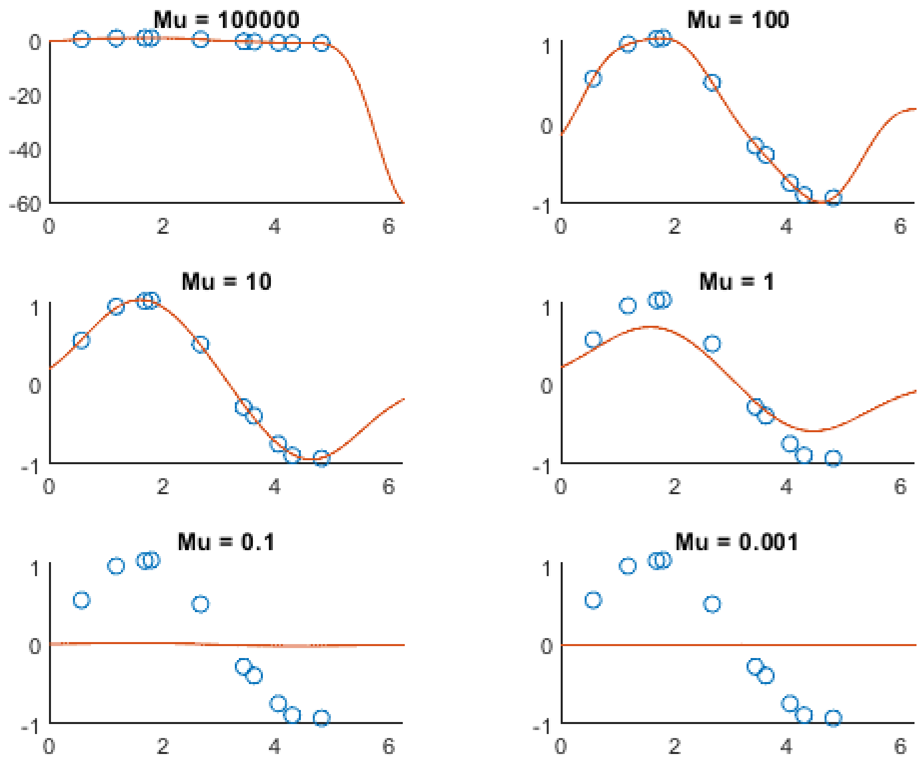
Part 6

Do the same for other values of the hyperparameter such as μ = 10 and μ = 1. **

I notice that as mu becomes small the function gets a lot more regularized. So at high values of mu the function is overfit, while at low values of mu it is underfit and becomes a line.

```
mus = [10^5, 100, 10, 1, 0.1, 0.001];
figure()
tiledlayout(3,2)
for mu = mus
    params = gauss_basis(0, 2 * pi, 10);
    [~, M] = size(params);
    func = @func_gauss;
    w = lsefit(x, t, params, func, mu);

    x_test = (0:0.01:2* pi)';  % Sample points to look at function
    [N, ~] = size(x_test);
    Sig_test =  eval_basis(params, func, x_test);
```

```
    y = Sig_test * w;

    nexttile
    hold on
    scatter(x, t);
    plot(x_test, y);
    title("Mu = " + mu);
    hold off
end
```



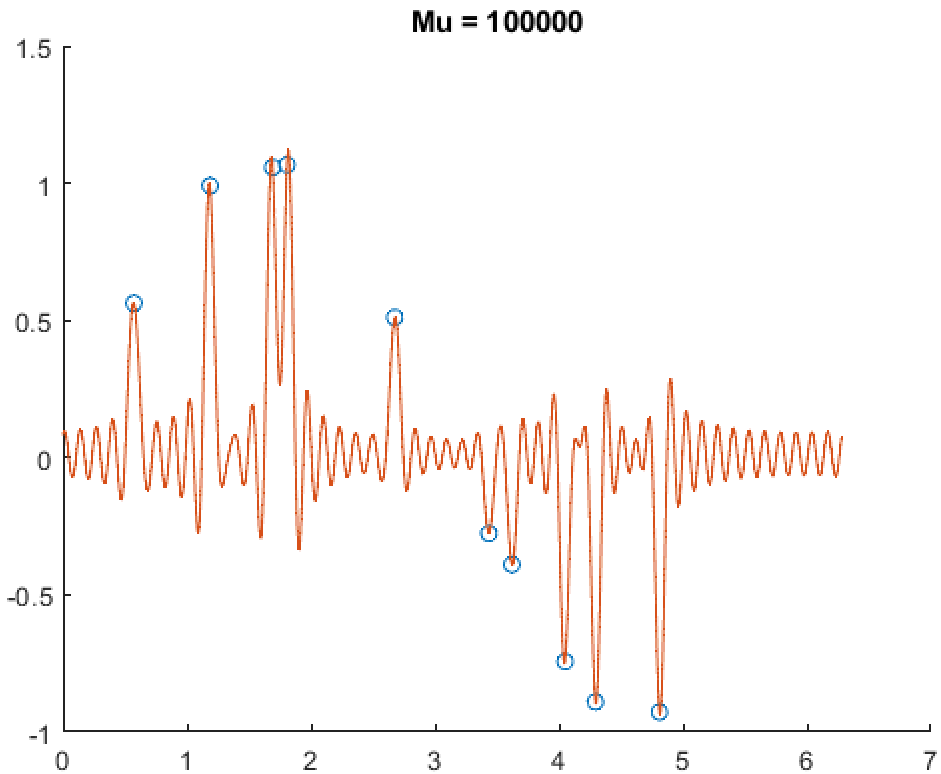
Extra 2

Train the model on the data in simple.mat using ten **fourier** functions and $\mu = 105$. Plot and turn in the learned model (the function fit to the data) on the interval $[0,2\pi]$.
****

Note, I set M to be big here as I thought it was very fun.

```
params = fourier_basis(0, 2 * pi, 100);
[~, M] = size(params);
func = @func_fourier;

mu = 10^(5);
w = lsefit(x, t, params, func, mu);

x_test = (0:0.001:2* pi)';  % Sample points to look at function
```

```
[N, ~] = size(x_test);
Sig_test =  eval_basis(params, func, x_test);

y = Sig_test * w;

figure();
hold on
scatter(x, t);
plot(x_test, y);
title("Mu = " + mu);
hold off
```



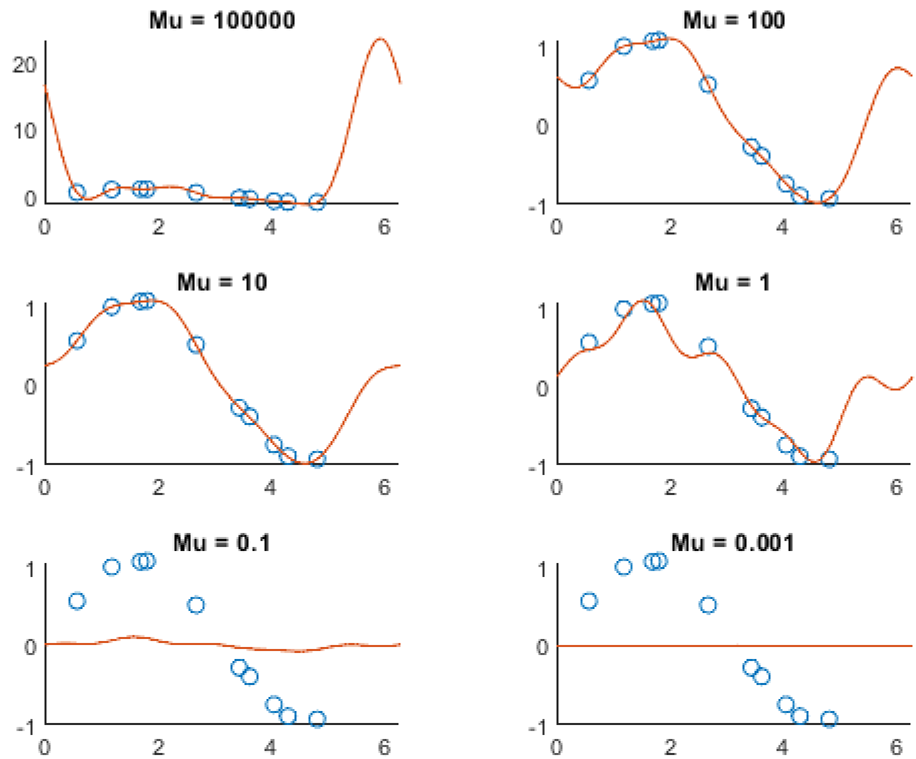
Extra 3

Do the same for other values of the hyperparameter such as μ = 10 and μ = 1.
****

I notice that as mu becomes small the function gets a lot more regularized. So at high values of mu the function is overfit, while at low values of mu it is underfit and becomes a line. The overfitting is a lot more obvious in Extra 2 than in the others. It is pretty cool.

```
mus = [10^5, 100, 10, 1, 0.1, 0.001];
figure()
tiledlayout(3,2)
for mu = mus
    params = fourier_basis(0, 2 * pi, 10);
    [~, M] = size(params);
```

```
    func = @func_fourier;
    w = lsefit(x, t, params, func, mu);

    x_test = (0:0.01:2* pi)';  % Sample points to look at function
    [N, ~] = size(x_test);
    Sig_test =  eval_basis(params, func, x_test);

    y = Sig_test * w;

    nexttile
    hold on
    scatter(x, t);
    plot(x_test, y);
    title("Mu = " + mu);
    hold off
end
```



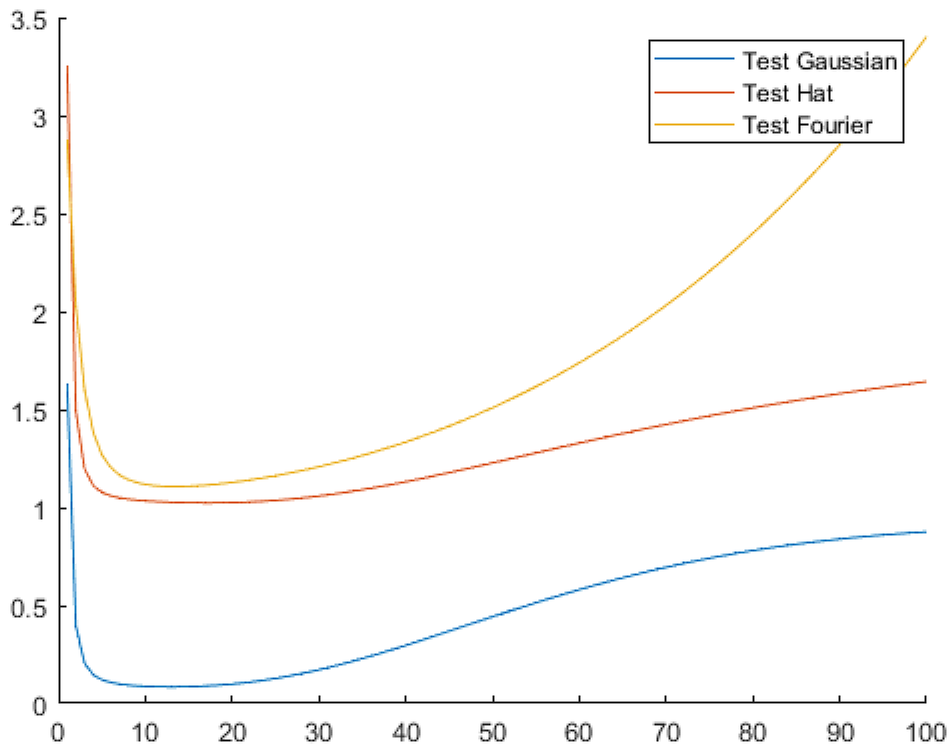
Part 8

Fit the model with integer values of μ = 1 to 100, and using a Gaussian basis of M = 10 elements on the data in simple.mat. For each model, calculate the squared error for the observations in test.mat (therefore testing the model). Generate and turn in a plot with μ on the x-axis and the total squared model error on the test data along the y-axis

```
gauss_params = gauss_basis(0, 2 * pi, 10);
gauss_func = @func_gauss;
```

```
hat_params = hat_basis(0, 2 * pi, 10);
hat_func = @func_hat;

four_params = fourier_basis(0, 2 * pi, 10);
four_func = @func_fourier;

load("test.mat")

e_g = zeros(1, 100);
e_h = zeros(1, 100);
e_f = zeros(1, 100);

t_g = zeros(1, 100);
t_h = zeros(1, 100);
t_f = zeros(1, 100);

for mu = 1:100
    w_g = lsefit(x, t, gauss_params, gauss_func, mu);
    w_h = lsefit(x, t, hat_params, hat_func, mu);
    w_f = lsefit(x, t, four_params, four_func, mu);

    sig_g = eval_basis(gauss_params, gauss_func, test_x);
    sig_h = eval_basis(hat_params, hat_func, test_x);
    sig_f = eval_basis(four_params, four_func, test_x);

    y_g = sig_g * w_g;
    y_h = sig_h * w_h;
    y_f = sig_f * w_f;

    e_g(mu) = sum((y_g - test_t).^2);
    e_h(mu) = sum((y_h - test_t).^2);
    e_f(mu) = sum((y_f - test_t).^2);

    sig_g2 = eval_basis(gauss_params, gauss_func, x);
    sig_h2 = eval_basis(hat_params, hat_func, x);
    sig_f2 = eval_basis(four_params, four_func, x);

    y_g = sig_g2 * w_g;
    y_h = sig_h2 * w_h;
    y_f = sig_f2 * w_f;

    t_g(mu) = sum((y_g - t).^2);
    t_h(mu) = sum((y_h - t).^2);
    t_f(mu) = sum((y_f - t).^2);
end

x_axis = [1:100];
figure()
hold on
plot(x_axis, e_g);
plot(x_axis, e_h);
plot(x_axis, e_f);
```

```
% plot(x_axis, t_g);
% plot(x_axis, t_h);
% plot(x_axis, t_f);
hold off

legend('Test Gaussian','Test Hat', 'Test Fourier')
```



Part 9

What value of μ, when trained on the data in simple.mat, performs best on the data in test.mat? How do you know? Explain the shape of the plot you generated in the previous step. **

Below we print the optimal mu value. I know it is optimal (for the test set) becaus it is the mu that minimizes the squared error on the test set. The shape is interesting, to the left of the minimum the model is underfit and has not learned the data. And to the right the model is overfitting to the training data, making it perform worse on the test data.

```
[M,optimal_mu] = min(e_g);
optimal_mu


optimal_mu =

    13
```

Part 10

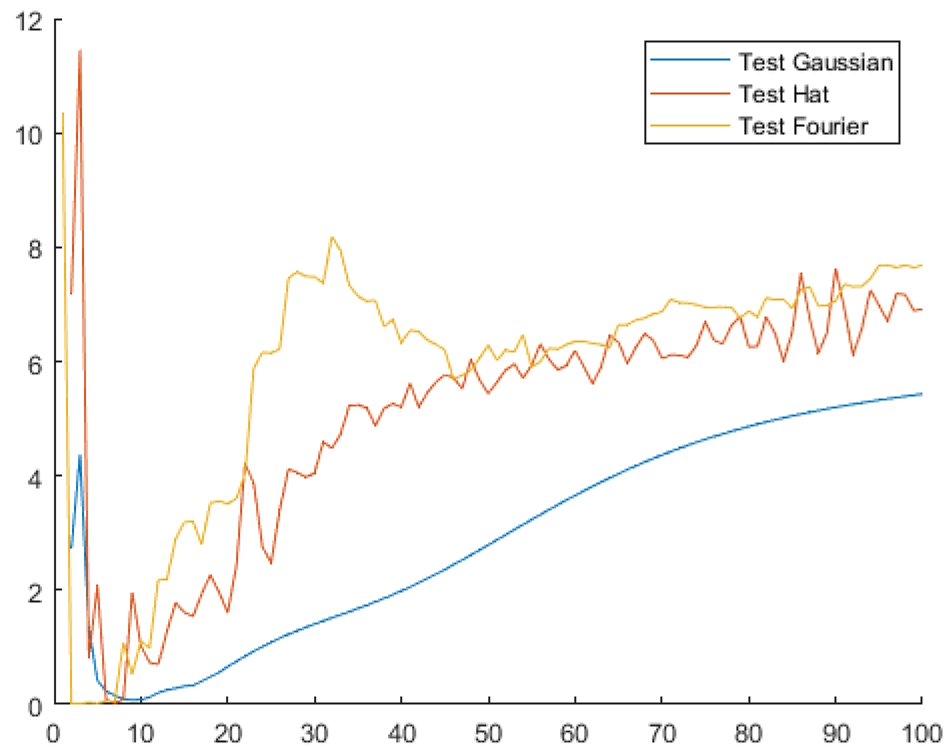
Repeat the process, now fixing μ = 13 and varying the number of basis elements from M = 1 to 100. Generate and turn in a plot with the number of basis elements on the x-axis and the error for the test data on the y-axis **

This is also intereresting. We see a similar phenomenon as with the mu. Which is cool because it isn't the same as regularization which is underfitting and overfitting with a term, but it is the propper amount of basis functions. Essentially too few basis functions and it can't express the target function fully, but too few basis functions and the model gets too flexible. The optimal M is 9. It is also cool how the fourier and hat basis wiggle a lot over M. I don't have any good ideas as to why this happens.

```
load("test.mat")

e_g = zeros(1, 100);
e_h = zeros(1, 100);
e_f = zeros(1, 100);

t_g = zeros(1, 100);
t_h = zeros(1, 100);
t_f = zeros(1, 100);

mu = 13;

for M = 1:100
    gauss_params = gauss_basis(0, 2 * pi, M);
    gauss_func = @func_gauss;

    hat_params = hat_basis(0, 2 * pi, M);
    hat_func = @func_hat;

    four_params = fourier_basis(0, 2 * pi, M);
    four_func = @func_fourier;

    w_g = lsefit(x, t, gauss_params, gauss_func, mu);
    w_h = lsefit(x, t, hat_params, hat_func, mu);
    w_f = lsefit(x, t, four_params, four_func, mu);

    sig_g = eval_basis(gauss_params, gauss_func, test_x);
    sig_h = eval_basis(hat_params, hat_func, test_x);
    sig_f = eval_basis(four_params, four_func, test_x);

    y_g = sig_g * w_g;
    y_h = sig_h * w_h;
    y_f = sig_f * w_f;

    e_g(M) = sum((y_g - test_t).^2);
    e_h(M) = sum((y_h - test_t).^2);
    e_f(M) = sum((y_f - test_t).^2);

    sig_g2 = eval_basis(gauss_params, gauss_func, x);
    sig_h2 = eval_basis(hat_params, hat_func, x);
    sig_f2 = eval_basis(four_params, four_func, x);
```

```
    y_g = sig_g2 * w_g;
    y_h = sig_h2 * w_h;
    y_f = sig_f2 * w_f;

    t_g(M) = sum((y_g - t).^2);
    t_h(M) = sum((y_h - t).^2);
    t_f(M) = sum((y_f - t).^2);
end

x_axis = [1:100];
figure()
hold on
plot(x_axis, e_g);
plot(x_axis, e_h);
plot(x_axis, e_f);

% plot(x_axis, t_g);
% plot(x_axis, t_h);
% plot(x_axis, t_f);
hold off
legend('Test Gaussian','Test Hat', 'Test Fourier')

[M,optimal_M] = min(e_g);
optimal_M
```

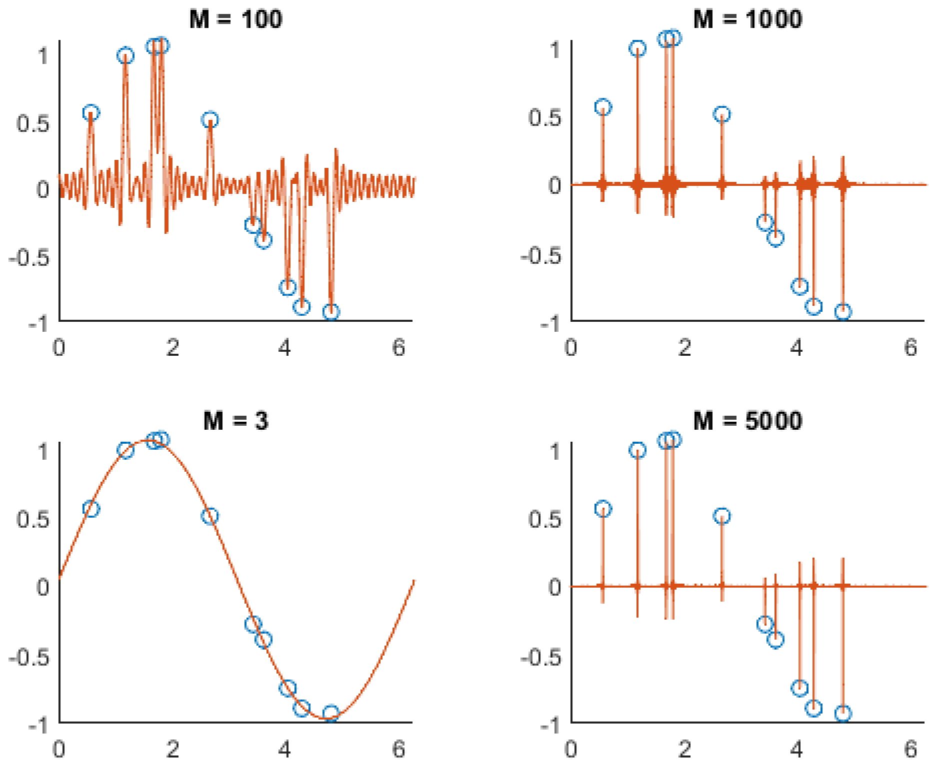
```
optimal_M =

     9
```



Kicks and Giggles

```
figure();
tiledlayout(2,2)

func = @func_fourier;
mu = 10^(5);

params = fourier_basis(0, 2 * pi, 100);
[~, M] = size(params);
w = lsefit(x, t, params, func, mu);

x_test = (0:0.001:2* pi)';  % Sample points to look at function
[N, ~] = size(x_test);
Sig_test =  eval_basis(params, func, x_test);

y = Sig_test * w;

nexttile
hold on
scatter(x, t);
plot(x_test, y);
title("M = " + M);
hold off
```

```
params = fourier_basis(0, 2 * pi, 1000);
[~, M] = size(params);
w = lsefit(x, t, params, func, mu);

x_test = (0:0.001:2* pi)';  % Sample points to look at function
[N, ~] = size(x_test);
Sig_test =  eval_basis(params, func, x_test);

y = Sig_test * w;

nexttile
hold on
scatter(x, t);
plot(x_test, y);
title("M = " + M);
hold off

params = fourier_basis(0, 2 * pi, 3);
[~, M] = size(params);
w = lsefit(x, t, params, func, mu);

x_test = (0:0.001:2* pi)';  % Sample points to look at function
[N, ~] = size(x_test);
Sig_test =  eval_basis(params, func, x_test);

y = Sig_test * w;

nexttile
hold on
scatter(x, t);
plot(x_test, y);
title("M = " + M);
hold off

params = fourier_basis(0, 2 * pi, 5000);
[~, M] = size(params);
w = lsefit(x, t, params, func, mu);

x_test = (0:0.0001:2* pi)';  % Sample points to look at function
[N, ~] = size(x_test);
Sig_test =  eval_basis(params, func, x_test);

y = Sig_test * w;

nexttile
hold on
scatter(x, t);
plot(x_test, y);
title("M = " + M);
hold off
```



lsefit.m

```
%% Least-Squared Error FIT
%  Find the linear combination of basis functions which best model the
 data.
%
%  Inputs:
%
%  x - Vector with observation locations in 1D. (indep. variable)
%  t - Vector with observations in 1D. (dep. variable)
%  params - Parameters for the basis functions to be used in func,
 e.g. as
%    produced by gauss_basis.
%  func - Function handle which evaluates a basis function with
 parameters
%    given by the columns of params and at the specified locations.
 e.g.
%    @gauss_basis, or @hat_basis.
%    For example, the first basis function at x = 2 is func(2,
 params(:,1)).
%  mu - Scalar representing the standard deviation of the prior
 Gaussian on
%    the model parameters.
%
```

```
%  Outputs:
%  w - Coefficients used to generate a linear combination of the
 basis
%    functions which is the maximum likelihood learned model.

function [w] = lsefit(x, t, params, func, mu)
    % wmle = inv(Sig'*Sig + 1/mu^2 I) Sig' t
    [N, ~] = size(x);
    [~, M] = size(params);

    % Compute Sigma
    Sig = eval_basis(params, func, x);

    Z = Sig' * Sig + 1/mu^2 * eye(M);
    w = Z \ (Sig' * t);
end
```

hat_basis.m

```
%% generate a HAT BASIS of functions.
%%  Produces parameters for a 1D basis of hat functions on an
 interval.
%% Inputs:
%%  a - Beginning of the interval.
%%  b - End of the interval.
%%  num - Number of elements to generate.
%% Outputs:
%%  params - Matrix with, in each column, the parameters of a basis
 element.

function [params] = hat_basis(a, b, num)
    params = zeros(2, num);
    spacing = (b - a)/(num - 1);
    for n = 1:num
        c = a + (n - 1)*spacing;
        params(1,n) = c - spacing;
        params(2,n) = c + spacing;
    end
end
```

func_hat.m

```
function [v] = func_hat(x, params)
    c = 0.5*(params(1) + params(2));
    v = (x < c) .* (x - params(1))./(c - params(1));
    v = v + (x >= c) .* (1 - (x - c)./(params(2) - c));
    v(x < params(1)) = 0;
    v(x > params(2)) = 0;
end
```

gauss_basis.m

```
%% generate a GAUSSian BASIS of functions.
%%  Produces parameters for a 1D basis of Gaussians on an interval.
%% Inputs:
%%  a - Beginning of the interval.
%%  b - End of the interval.
%%  num - Number of elements to generate.
%% Outputs:
%%  params - Matrix with, in each column, the parameters of a basis
 element.

function [params] = gauss_basis(a, b, num)
    params = zeros(2, num);
    for n = 1:num
        params(1,n) = a + (n - 1)*(b - a)/(num - 1);
        params(2,n) = (b-a)/num;
    end
end
```

func_gauss.m

```
function [v] = func_gauss(x, params)
    v = (1/(params(2)*sqrt(2*pi))) .* exp(-(x - params(1)).^2 ./
 (2*params(2)^2));
end
```

fourier_basis.m

```
function [params] = fourier_basis(a, b, num)
%FOURIER_BASIS Summary of this function goes here
%   Detailed explanation goes here
    params = zeros(3, num);  % sin(k(x-a) + b)
    k = (2 * pi) / (b - a);  % so it covers full range a,b
    offset = (b - a + pi/2) / (2 * pi);  % offset for cos functions
    scalar = 0;
    for n = 1:num
        params(1,n) = scalar * k;
        params(2,n) = a;
        if mod(n, 2) == 0
            params(3,n) = 0;
        else
            scalar = scalar + 1;
            params(3,n) = pi/2;
        end
```

```
    end
end
```

func_fourier.m

```
function [v] = func_fourier(x, params)
    k = params(1);
    a = params(2);
    b = params(3);
    v = sin(k * (x - a) + b);
end
```

eval_basis.m

```
%% EVALuate BASIS functions
%%  Calculate the values of a collection of basis functions at the
 specified places.
%% Inputs:
%%  params - Matrix with, in each column, the parameters for a basis
 function.
%%  func - Function handle which, when combined with the parameters,
 calculates
%%    the value of a basis function element.
%%  xeval - X-coordinates at which each basis function is evaluated.
%% Outputs:
%%  B - Matrix with the values of the basis functions at the locations
 in xeval.
%%    Each column of B corresponds to a basis function.

function [B] = eval_basis(params, func, xeval)
    B = zeros(length(xeval), size(params,2));
    for j = 1:size(params,2)
        B(:,j) = func(xeval, params(:,j));
    end
end
```

Published with MATLAB® R2021a