This week we discussed singular value decomposition (SVD) and how to best use it in our favorite problem, regression. SVD is a very powerful technique for decomposing a matrix into orthogonal and diagonal matrices. This allows us to quickly solve least squares problems as well as analyze the original matrix with respect to the problem. In least squares we need to calculate the pseudoinverse of A and that is easily as follows:
A = U \Sigma V.T
A\dagger = V \Sigma^(-1) U.T,
simplifying the computation.

The SVD decomposition also gives us the ability to analyze singular values and truncate some of our "data" so as to achieve a better generalization fit and not to overfit. This is done by determining which singular values are too small and zeroing them, so that the inverse does not blow up, while still maintaining accuracy.

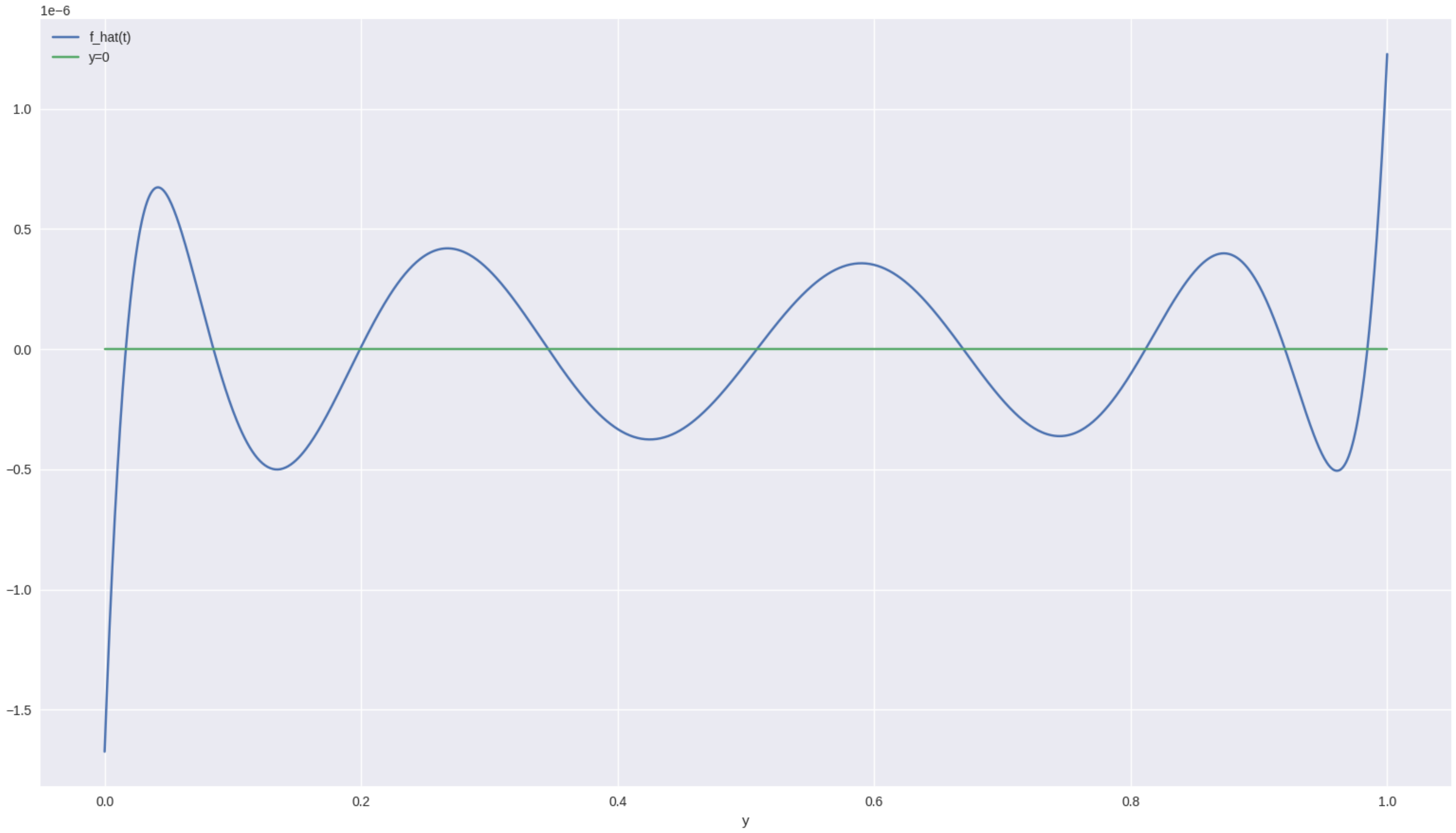
We also covered iterative methods like gradient descent and conjugate gradients. Gradient descent follows the direction of the largest gradient, while conjugate gradients takes orthogonal based steps to iteratively minimize the error in each direction.

These method covered above give us robust and practical regression method to use in a variety of situations. In fact methods like gradient descent and conjugate gradients are used as popular optimizers for deep learning frameworks.

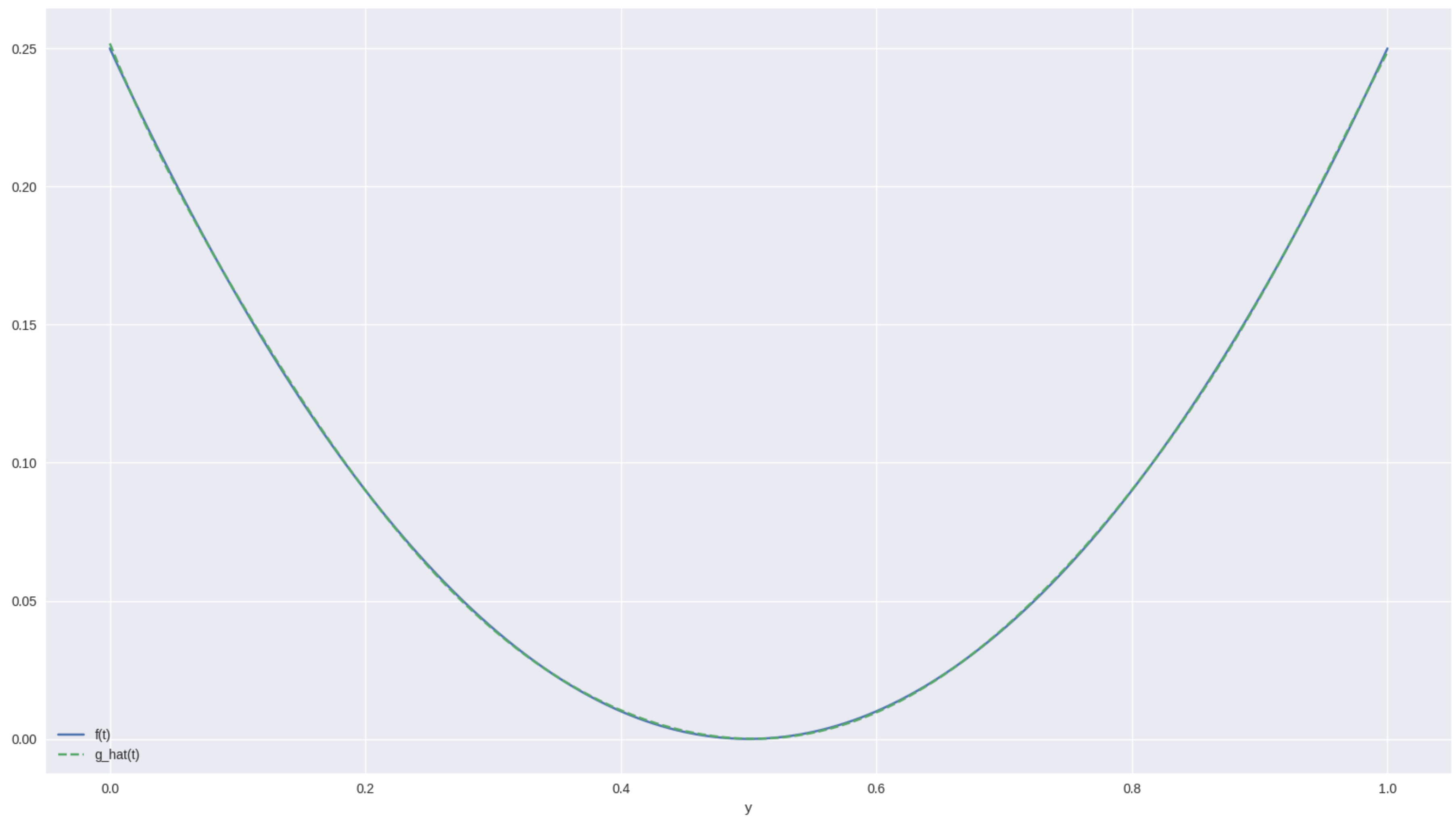
```python
"""Problem 2."""

import matplotlib as mpl
import matplotlib.pyplot as plt

import numpy as np

import scipy.integrate as integrate

mpl.style.use('seaborn')


def f_hat(_input, alpha_coeff_vec):
    """f_hat."""
    return np.polyval(np.flip(alpha_coeff_vec), _input)


def compute_poly_gram_matrix(basis_num):
    """Compute gram matrix."""
    gram_mat = np.zeros((basis_num, basis_num))

    def f_hat_i_f_hat_j(_input, i, j):
        f_hat_i_coeff = np.zeros(i + 1)
        f_hat_i_coeff[i] = 1
        f_hat_j_coeff = np.zeros(j + 1)
        f_hat_j_coeff[j] = 1
        return f_hat(_input, f_hat_i_coeff) * f_hat(_input, f_hat_j_coeff)

    for i in range(0, basis_num):
        for j in range(0, basis_num):
            gram_mat[i, j] = integrate.quad(
                f_hat_i_f_hat_j, 0, 1, args=(i, j))[0]

    return gram_mat


def part_a():
    """Part a."""
    print('Part a')

    order = 9
    basis_num = order + 1
    gram_mat = compute_poly_gram_matrix(basis_num)

    eigenvalue_vec, eigenvector_mat = np.linalg.eig(gram_mat)
    min_eigenvalue_idx = eigenvalue_vec.argmin()

    alpha_vec = eigenvector_mat[:, min_eigenvalue_idx]

    print('alpha_vec: ' + str(alpha_vec))

    fig = plt.figure()
    fig.suptitle('f_hat(t)')
    axes = fig.add_subplot(111)

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

    axes.plot(t_axes, f_hat(t_axes, alpha_vec), label='f_hat(t)')
    axes.plot(t_axes, np.zeros(t_axes.shape), label='y=0')

    axes.set_xlabel('t')
    axes.set_xlabel('y')
    axes.legend()
    plt.show()

    return gram_mat


gram_mat_part_a = part_a()


def part_b(gram_mat):
    """Part b."""
    print('Part b')

    order = 9
    basis_num = order + 1
    alpha_vec = np.zeros(basis_num)
    alpha_vec[0] = 1/4
    alpha_vec[1] = -1
    alpha_vec[2] = 1

    max_approx_error = 1e-6
    min_coeff_norm_sqrd = 1e6

    # int_0^1 (f(t) - g(t))**2 dt
    # f(t) - g(t) = p(t)
    # int_0^1 (p(t))**2 dt // gamma_vec is coeff of p(t)
    # gamma_vec.T @ gram_mat @ gamma_vec
    # np.transpose(np.transpose(eigenvector_mat) @ gamma_vec) @
    # np.diag(eigenvalue_vec) @ (np.transpose(eigenvector_mat) @ gamma_vec)
    # Let delta_vec = np.transpose(eigenvector_mat) @ gamma_vec (73)
    # sum_i=0^9 (eigenvalue_vec[i] * delta_vec[i]**2)

    # sum_i=0^9 (alpha_vec_n - beta_vec_n)**2
    # sum_i=0^9 (gamma_vec_n)**2
    # np.transpose(gamma_vec) @ gamma_vec
    # (73) => gamma_vec = eigenvector_mat @ delta_vec
    # np.transpose(eigenvector_mat @ delta_vec) @ eigenvector_mat @ delta_vec
    # np.transpose(delta_vec) @ delta_vec

    eigenvalue_vec, eigenvector_mat = np.linalg.eig(gram_mat)
    min_eigenvalue_idx = eigenvalue_vec.argmin()

    delta_vec = np.zeros(basis_num)
    delta_vec[min_eigenvalue_idx] = (1 + 1e-1) * np.sqrt(min_coeff_norm_sqrd)

    # Check two conditions
    # sum_i=0^9 (eigenvalue_vec[i] * delta_vec[i]**2) < max_approx_error
    # np.transpose(delta_vec) @ delta_vec > min_coeff_norm_sqrd

    approx_error = sum((eigenvalue_vec[i] * delta_vec[i] ** 2 for i in
                        range(len(eigenvalue_vec))))
    if approx_error < max_approx_error:
        print('approx_error < max_approx_error')

    coeff_norm_error = np.transpose(delta_vec) @ delta_vec
    if coeff_norm_error > min_coeff_norm_sqrd:
        print('coeff_norm_error > min_coeff_norm_error')

    gamma_vec = eigenvector_mat @ delta_vec
    beta_vec = alpha_vec - gamma_vec

    print('beta_vec: ' + str(beta_vec))

    fig = plt.figure()
    fig.suptitle('g_hat(t)')
    axes = fig.add_subplot(111)

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

    axes.plot(t_axes, f_hat(t_axes, alpha_vec), label='f(t)')
    axes.plot(t_axes, f_hat(t_axes, beta_vec), '--', label='g_hat(t)')

    axes.set_xlabel('t')
    axes.set_xlabel('y')
    axes.legend()

    plt.show()


part_b(gram_mat_part_a)
```

```
Part a
alpha_vec: [-1.67404617e-06  1.45568480e-04 -3.11589248e-03  2.84376369e-02
 -1.36071670e-01  3.75033997e-01 -6.16652565e-01  5.97017357e-01
 -3.13921241e-01  6.91297093e-02]
Part b
approx_error < max_approx_error
coeff_norm_error > min_coeff_norm_error
beta_vec: [ 2.51841451e-01 -1.16012533e+00  4.42748172e+00 -3.12814006e+01
  1.49678836e+02 -4.12537396e+02  6.78317821e+02 -6.56719093e+02
  3.45313365e+02 -7.60426802e+01]
```

f_hat(t)



g_hat(t)



```python
"""Problem 3."""
import matplotlib as mpl
import matplotlib.pyplot as plt

import numpy as np

import scipy.integrate as integrate
import scipy.io as sio
import scipy.special as ssp

mpl.style.use('seaborn')

MAT_FILENAME = 'hw06p3_clusterdata.mat'
data_samples = sio.loadmat(MAT_FILENAME)

t_data = data_samples['T']
y_data = data_samples['y']


def f_true(_input):
    """f_true."""
    return (np.sin(12 * (_input + 0.2)))/(_input + 0.2)


def l_n(_input, order):
    """Legendre."""
    return np.polyval(ssp.legendre(order), _input)


def v_n(_input, order):
    """Normalize Legendre."""
    return np.sqrt(2) * np.sqrt((2 * order + 1)/2) * l_n(2 * _input - 1, order)


def f_hat(_input, w_coeff):
    """f_hat."""
    return sum((w_coeff[idx] * v_n(_input, idx)
                for idx in range(len(w_coeff))))


def compute_legendre_fit(t_vec, y_vec, order):
    """Compute legendre fit."""
    basis_num = order + 1
    a_mat = np.zeros((len(t_vec), basis_num))

    for row_idx in range(len(t_vec)):
        a_mat[row_idx, :] = [v_n(t_vec[row_idx], order) for order in
                             range(basis_num)]

    w_coeff = np.linalg.inv(np.transpose(
        a_mat) @ a_mat) @ np.transpose(a_mat) @ y_vec

    return a_mat, w_coeff


def compute_sample_error(y_vec, a_mat, w_coeff):
    """Compute sample error."""
    sample_error = np.linalg.norm(y_vec - a_mat @ w_coeff)
    return sample_error


def plot_prediction_over_data(t_vec, y_vec, w_coeff, delta=0):
    """Plot f_true, f_data, f_pred."""
    fig = plt.figure()
    title = 'f_hat'
    if delta != 0:
        title += ' delta=' + str(delta)
    fig.suptitle(title)
    axes = fig.add_subplot(111)

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

    axes.plot(t_axis, f_true(t_axis), label='f_true(t)')
    axes.scatter(t_vec, y_vec, label='y_vec')
    axes.plot(t_axis, f_hat(t_axis, w_coeff), label='f_hat(t) ' +
              str(len(w_coeff) - 1) + ' order fit')

    axes.set_xlabel('t')
    axes.set_ylabel('y')
    axes.legend()

    plt.show()


def part_a(t_vec, y_vec):
    """Part a."""
    print('Part a')

    order = 3
    a_mat, w_coeff = compute_legendre_fit(t_vec, y_vec, order)
    print('w_coeff: ' + str(w_coeff))

    sample_error = compute_sample_error(y_vec, a_mat, w_coeff)
    print('sample_error: ' + str(sample_error))

    plot_prediction_over_data(t_vec, y_vec, w_coeff)

    return w_coeff


w_coeff_part_a = part_a(t_data, y_data)


def compute_generalization_error(w_coeff):
    """Compute generalization error."""

    def f_hat_f_true(_input, w_coeff):
        return (f_hat(_input, w_coeff) - f_true(_input)) ** 2

    generalization_error = np.sqrt(integrate.quad(f_hat_f_true, 0, 1,
                                                  args=(w_coeff))[0])
    return generalization_error


def part_b(w_coeff):
    """Part b."""
    print('Part b')

    generalization_error = compute_generalization_error(w_coeff)
    print('generalization_error: ' + str(generalization_error))


part_b(w_coeff_part_a)


def min_max_singular_value(a_mat):
    """Find minimum and maximum singular values."""
    _, singular_values, _ = np.linalg.svd(a_mat)
    smallest_singular_value = singular_values[0]
    largest_singular_value = singular_values[-1]
    return [smallest_singular_value, largest_singular_value]


def plot_error(error_list, domain_list, x_label, y_label):
    """Plot error."""
    fig = plt.figure()
    title = y_label + ' error'
    fig.suptitle(title)
    fig.suptitle(title)
    axes = fig.add_subplot(111)

    axes.scatter(domain_list, error_list)

    if x_label == 'delta':
        axes.set_xscale('log')

    axes.set_xlabel(x_label)
    axes.set_ylabel('error')

    plt.show()


def part_c(t_vec, y_vec):
    """Part c."""
    print('Part c')

    poly_order_list = [5, 10, 15, 20, 25]
    basis_num_list = [poly_order + 1 for poly_order in poly_order_list]

    a_mat_list = [None] * len(basis_num_list)
    w_coeff_list = [None] * len(basis_num_list)
    sample_error_list = [0] * len(basis_num_list)
    generalization_error_list = [0] * len(basis_num_list)

    for basis_num_index in range(len(basis_num_list)):
        poly_order = poly_order_list[basis_num_index]
        print('Polynomial Order: ' + str(poly_order))

        a_mat_list[basis_num_index], w_coeff_list[basis_num_index] = \
            compute_legendre_fit(t_vec, y_vec, poly_order)
        print('w_coeff: ' + str(w_coeff_list[basis_num_index]))

        sample_error_list[basis_num_index] = \
            compute_sample_error(y_vec,
                                 a_mat_list[basis_num_index],
                                 w_coeff_list[basis_num_index])
        print('sample_error: ' + str(sample_error_list[basis_num_index]))

        generalization_error_list[basis_num_index] = \
            compute_generalization_error(w_coeff_list[basis_num_index])
        print('generalization error: ' +
              str(generalization_error_list[basis_num_index]))

        smallest_singular_value, largest_singular_value = \
            min_max_singular_value(a_mat_list[basis_num_index])
        print('smallest singular value: ' + str(smallest_singular_value))
        print('largest singular value: ' + str(largest_singular_value))

    plot_error(sample_error_list, basis_num_list, 'basis_num', 'sample')

    plot_error(generalization_error_list, basis_num_list, 'basis_num',
               'generalization')
    print("""
    Least squares starts to fall apart when the basis_num is 16 because at this
    polynomial order we end up fitting our sample data points well, but lose
    out on accuracy in the domain where we do not have enough data. This is
    also the reason why sample error goes down monotonically as we increase
    polynomial order, but the generalization error increases.
    """)

    for poly_order_index, poly_order in enumerate(poly_order_list):
        w_coeff = w_coeff_list[poly_order_index]

        plot_prediction_over_data(t_vec, y_vec, w_coeff)


part_c(t_data, y_data)


def plot_singular_values(a_mat):
    """Plot singular values."""
    _, singular_values, _ = np.linalg.svd(a_mat)

    fig = plt.figure()
    fig.suptitle('singular values')
    axes = fig.add_subplot(111)

    axes.scatter(range(a_mat.shape[1]), singular_values)

    axes.set_xlabel('basis_num')
    axes.set_ylabel('singular value')

    plt.show()


def compute_truncated_legendre_fit(y_vec, a_mat, r_prime):
    """Compute truncated legendre fit."""
    u_mat, s_vec, vh_mat = np.linalg.svd(a_mat)
    v_mat = np.transpose(vh_mat)
    s_mat = np.zeros((u_mat.shape[1], vh_mat.shape[0]))

    for i in range(min(s_mat.shape)):
        s_mat[i, i] = s_vec[i]

    truncated_s_mat = np.zeros(s_mat.shape)
    for i in range(r_prime):
        truncated_s_mat[i, i] = s_mat[i, i]

    truncated_s_inv_mat = np.transpose(np.zeros(truncated_s_mat.shape))
    for i in range(r_prime):
        truncated_s_inv_mat[i, i] = 1/truncated_s_mat[i, i]

    truncated_svd_inv_mat = v_mat @ truncated_s_inv_mat @ \
        np.transpose(u_mat)
    w_coeff = truncated_svd_inv_mat @ y_vec

    truncated_svd_mat = u_mat @ truncated_s_mat @ vh_mat

    return truncated_svd_mat, w_coeff, truncated_svd_inv_mat, u_mat, v_mat, \
        truncated_s_inv_mat


def part_d(t_vec, y_vec):
    """Part d."""
    print('Part d')
    basis_num = 25
    order = basis_num - 1

    a_mat, _ = compute_legendre_fit(t_vec, y_vec, order)

    plot_singular_values(a_mat)

    r_prime = 18
    truncated_svd_mat, w_coeff, _, _, _, _ = \
        compute_truncated_legendre_fit(y_vec, a_mat, r_prime)
    print("""
    r_prime was chosen to be 18 since based on our plot we have a sizeable drop
    in the magnitude of the singular values after 2, the number of values we
    have greater than or equal to 2 is 18.
    """)

    sample_error = compute_sample_error(y_vec, truncated_svd_mat, w_coeff)
    print('sample_error: ' + str(sample_error))

    generalization_error = compute_generalization_error(w_coeff)
    print('generalization error: ' + str(generalization_error))

    return a_mat


a_mat_part_d = part_d(t_data, y_data)


# pylint: disable=too-many-locals
def part_e(t_vec, y_vec, a_mat):
    """Part e."""
    print('Part e')

    basis_num = 25

    r_prime_list = list(range(5, basis_num))

    sample_error_list = [None] * len(r_prime_list)
    generalization_error_list = [None] * len(r_prime_list)
    noise_error_list = [None] * len(r_prime_list)
    approx_error_list = [None] * len(r_prime_list)
    null_space_error_list = [None] * len(r_prime_list)

    for r_prime_index, r_prime in enumerate(r_prime_list):
        truncated_svd_mat, w_coeff, truncated_svd_inv_mat, u_mat, v_mat, \
            truncated_s_inv_mat = \
            compute_truncated_legendre_fit(y_vec, a_mat, r_prime)

        sample_error_list[r_prime_index] = \
            compute_sample_error(y_vec, truncated_svd_mat, w_coeff)

        generalization_error_list[r_prime_index] = \
            compute_generalization_error(w_coeff)

        xo_vec = truncated_svd_inv_mat @ f_true(t_vec)
        xo_vec = xo_vec.flatten()
        error_vec = y_vec - f_true(t_vec)
        error_vec = error_vec.flatten()

        a_mat_rank = np.linalg.matrix_rank(a_mat)
        r_prime = np.linalg.matrix_rank(truncated_svd_mat)

        noise_error = 0
        for r_idx in list(range(a_mat_rank+1, basis_num)):
            noise_error += (1 / truncated_s_inv_mat[r_idx, r_idx]
                            * np.inner(error_vec, u_mat[:, r_idx]))**2
        noise_error = np.sqrt(noise_error)
        noise_error_list[r_prime_index] = noise_error

        approx_error = 0
        for r_idx in list(range(r_prime, a_mat_rank)):
            approx_error += (np.inner(xo_vec, v_mat[:, r_idx]))**2
        approx_error = np.sqrt(approx_error)
        approx_error_list[r_prime_index] = approx_error

        null_space_error = 0
        for r_idx in list(range(a_mat_rank, basis_num)):
            null_space_error += (np.inner(xo_vec, v_mat[:, r_idx]))**2
        null_space_error = np.sqrt(null_space_error)
        null_space_error_list[r_prime_index] = null_space_error

    plot_error(sample_error_list, r_prime_list, 'r_prime', 'sample')
    plot_error(generalization_error_list, r_prime_list,
               'r_prime', 'generalization')
    plot_error(noise_error_list, r_prime_list, 'r_prime', 'noise')
    plot_error(approx_error_list, r_prime_list, 'r_prime', 'approx')
    plot_error(null_space_error_list, r_prime_list, 'r_prime', 'null space')


part_e(t_data, y_data, a_mat_part_d)


def compute_ridge_legendre_fit(y_vec, a_mat, delta):
    """Compute ridge regression legendre fit."""
    u_mat, s_vec, vh_mat = np.linalg.svd(a_mat)
    v_mat = np.transpose(vh_mat)
    s_mat = np.zeros((u_mat.shape[1], vh_mat.shape[0]))

    for i in range(min(s_mat.shape)):
        s_mat[i, i] = s_vec[i]

    s_sqrd_mat = np.transpose(s_mat) @ s_mat

    w_coeff = v_mat @ \
        np.linalg.inv(s_sqrd_mat +
                      delta * np.identity(s_sqrd_mat.shape[0])) @ \
        np.transpose(s_mat) @ np.transpose(u_mat) @ y_vec

    return w_coeff


def part_f(t_vec, y_vec, a_mat):
    """Part f."""
    print('Part f')

    delta = 1e-5

    w_coeff = compute_ridge_legendre_fit(y_vec, a_mat, delta)

    plot_prediction_over_data(t_vec, y_vec, w_coeff, delta=delta)

    sample_error = compute_sample_error(y_vec, a_mat, w_coeff)
    print('sample_error: ' + str(sample_error))

    generalization_error = compute_generalization_error(w_coeff)
    print('generalization error: ' + str(generalization_error))
    delta_list = np.logspace(-6, 6, 13).tolist()

    sample_error_list = [None] * len(delta_list)
    generalization_error_list = [None] * len(delta_list)

    for delta_index, delta in enumerate(delta_list):

        w_coeff = compute_ridge_legendre_fit(y_vec, a_mat, delta)

        sample_error_list[delta_index] = \
            compute_sample_error(y_vec, a_mat, w_coeff)

        generalization_error_list[delta_index] = \
            compute_generalization_error(w_coeff)

    plot_error(sample_error_list, delta_list, 'delta', 'sample')
    plot_error(generalization_error_list,
               delta_list, 'delta', 'generalization')

    print("""
    As we sweep the value of delta the sample error is pretty stable until it
    increases at a certain inflection point. The generalization error on the
    other hand, starts high and then decreases before continuing to climb
    again.
    """)


part_f(t_data, y_data, a_mat_part_d)
```

```
Part a
w_coeff: [[ 0.61240722]
 [-0.46169933]
 [-0.05688734]
 [ 0.61338485]]
sample_error: 9.251142339796058
Part b
generalization_error: 1.7877793282737835
Part c
Polynomial Order: 5
w_coeff: [[ 0.20417535]
 [-0.0650089 ]
 [-0.03924705]
 [ 0.21481858]
 [ 0.77902869]
 [-0.65786507]]
sample_error: 1.5272594494759273
generalization error: 0.7172751484167387
smallest singular value: 15.57741953525218
largest singular value: 4.504648300787627
Polynomial Order: 10
w_coeff: [[-0.07050792]
 [ 0.21962486]
 [ 0.01615815]
 [-0.03944407]
 [ 0.94069675]
 [-0.65792181]
 [-0.21534148]
 [ 0.15852992]
 [ 0.04412051]
 [ 0.01384117]
 [-0.03475522]]
sample_error: 0.7700276242129308
generalization error: 0.16422847645666794
smallest singular value: 16.524708315036282
largest singular value: 0.5323015505974682
Polynomial Order: 15
w_coeff: [[-0.58091897]
 [ 0.67968187]
 [ 0.12604583]
 [-0.64569923]
 [ 1.35835377]
 [-0.4510216 ]
 [-0.66835704]
 [ 0.38924847]
 [ 0.14661786]
 [-0.2629238 ]
 [ 0.13894557]
 [ 0.06234942]
 [-0.17381424]
 [ 0.08649938]
 [ 0.07336213]
 [-0.04503685]]
sample_error: 0.720835743624414
generalization error: 1.075356905213417
smallest singular value: 17.340946868282707
largest singular value: 0.08714823187644298
Polynomial Order: 20
w_coeff: [[-1.84679443]
 [ 1.6737711 ]
 [ 0.54737814]
 [-1.9416046 ]
 [ 2.2120067 ]
 [-0.0691747 ]
 [-1.79398957]
 [ 1.08449555]
 [ 0.51175305]
 [-1.1366518 ]
 [ 0.54802099]
 [ 0.35220337]
 [-0.67859516]
 [ 0.33928622]
 [ 0.2104419 ]
 [-0.33279722]
 [ 0.12025354]
 [ 0.09442598]
 [-0.11929685]
 [ 0.02535131]
 [ 0.03391782]]
sample_error: 0.6973576051382525
generalization error: 4.005985627791585
smallest singular value: 18.663952639126414
largest singular value: 0.012588210551775147
Polynomial Order: 25
w_coeff: [[ 15.45507911]
 [-13.91893616]
 [ -2.83214439]
 [ 16.50076274]
 [-12.77625634]
 [ -2.12111495]
 [ 13.3284589 ]
 [-11.55835088]
 [ -0.09759784]
 [  9.71104144]
 [ -9.07676295]
 [  0.62966255]
 [  6.37589223]
 [ -5.99076144]
 [  0.68050818]
 [  3.54306487]
 [ -3.37179951]
 [  0.51193616]
 [  1.63891209]
 [ -1.63558294]
 [  0.29848207]
 [  0.69323925]
 [ -0.57345842]
 [  0.05763628]
 [  0.1857985 ]
 [ -0.1163034 ]]
sample_error: 0.6869148727136555
generalization error: 39.10527633402095
smallest singular value: 19.892037946902228
largest singular value: 0.0014654447026983194
```

Least squares starts to fall apart when the basis_num is 16 because at this polynomial order we end up fitting our sample data points well, but lose out on accuracy in the domain where we do not have enough data. This is also the reason why sample error goes down monotonically as we increase polynomial order, but the generalization error increases.

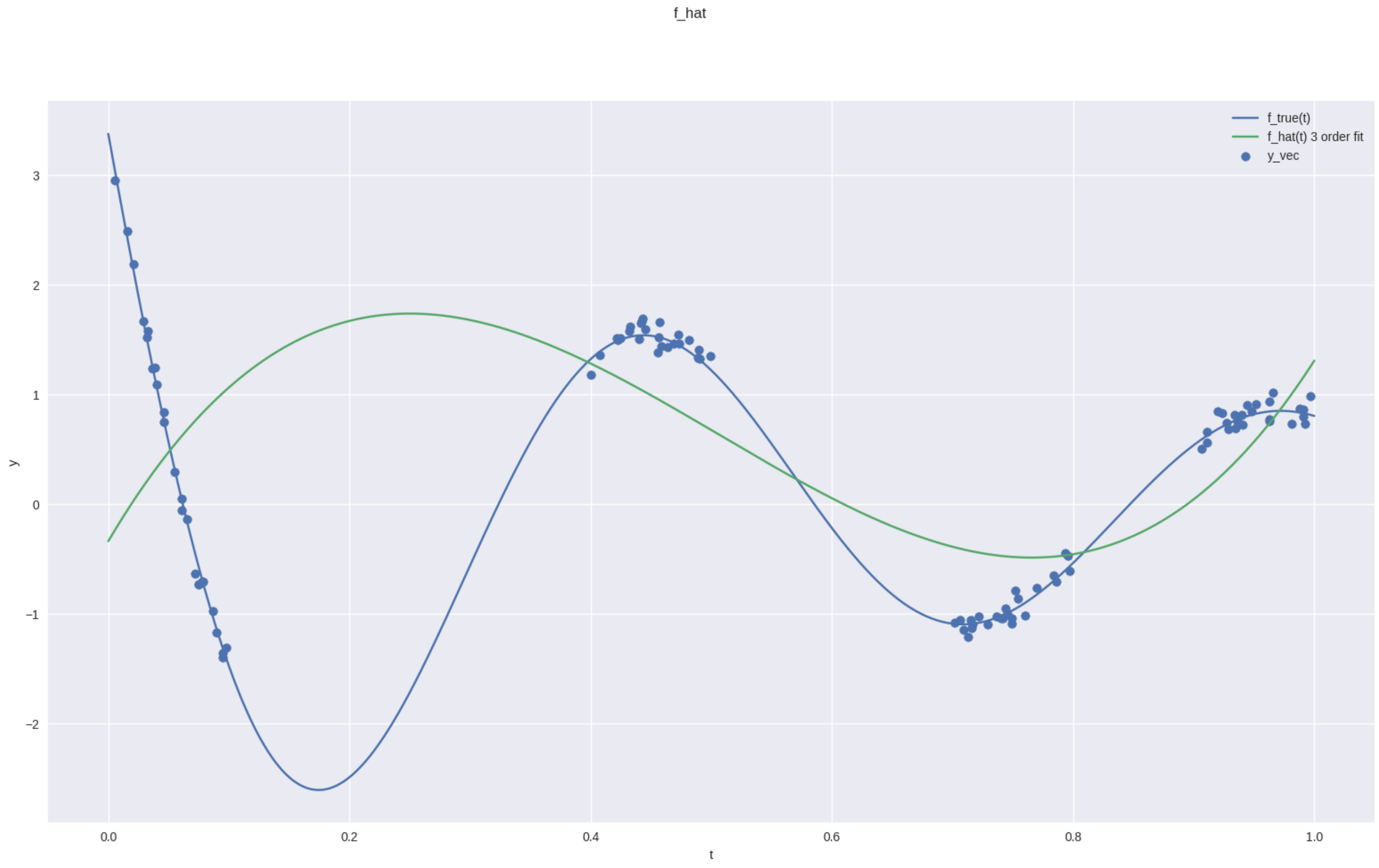
Part d

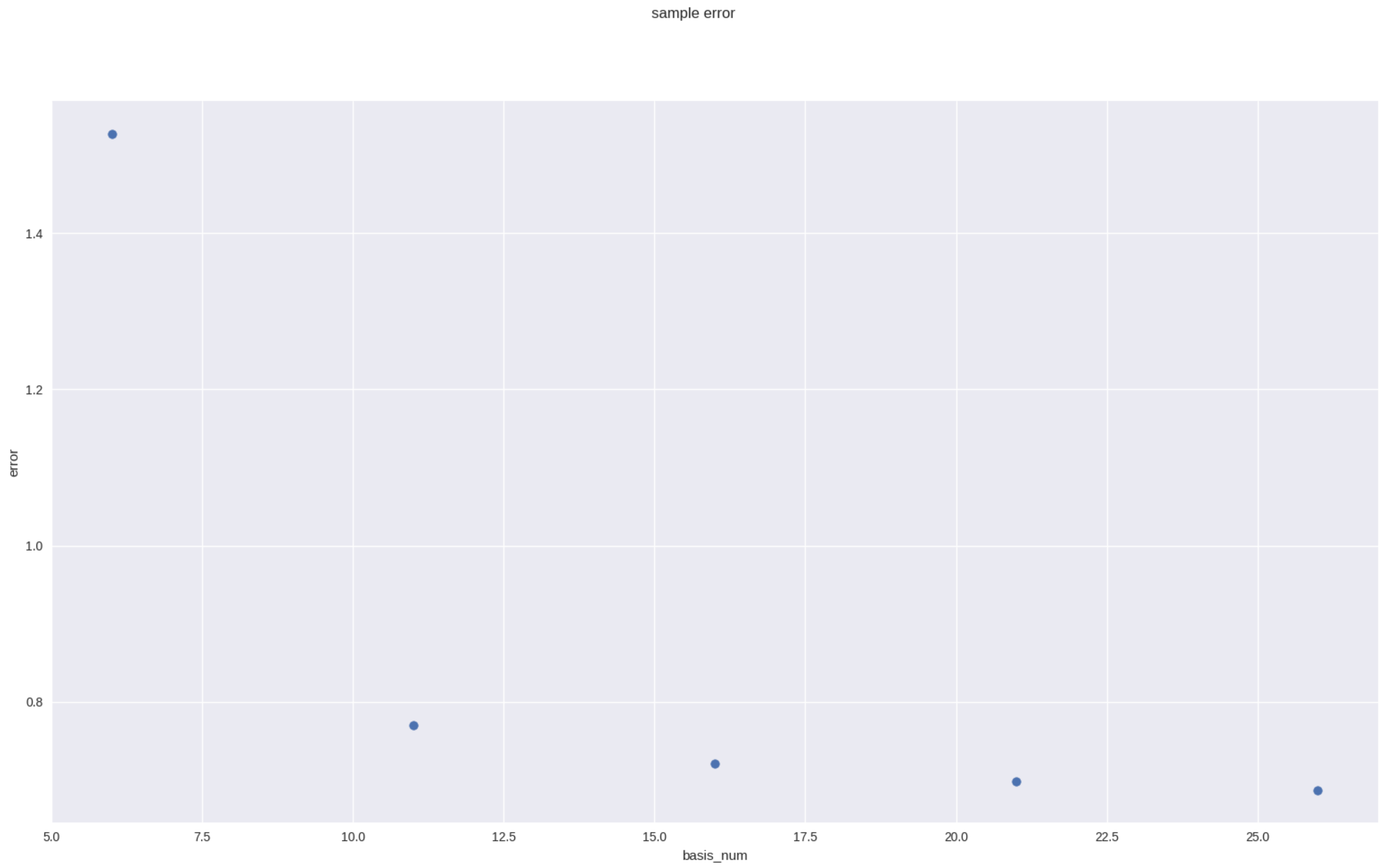
r_prime was chosen to be 18 since based on our plot we have a sizeable drop in the magnitude of the singular values after 2, the number of values we have greater than or equal to 2 is 18.

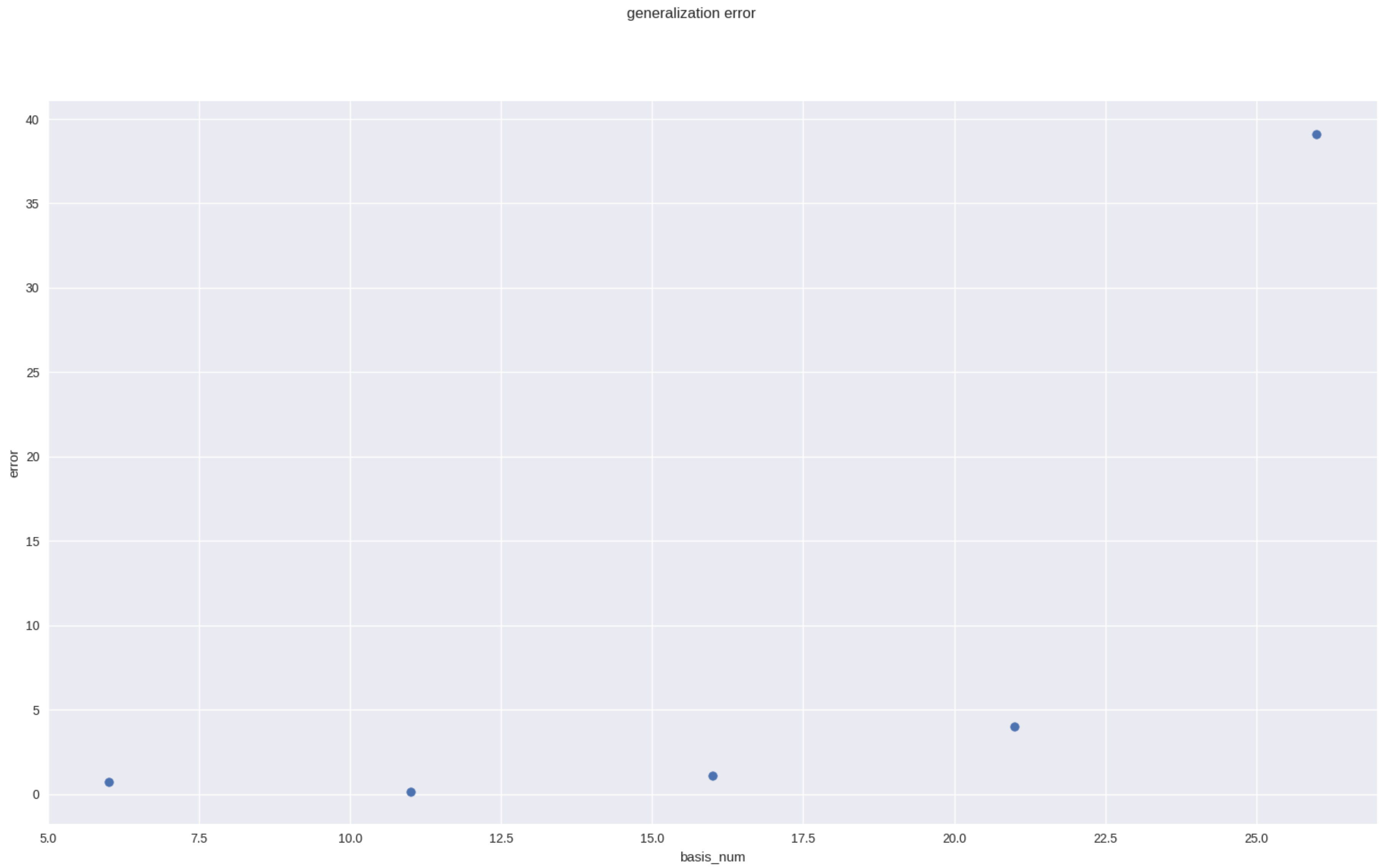
```
sample_error: 0.8013443167451153
```

```
generalization error: 0.9343561945039618
Part e
Part f
sample_error: 0.6902454173855826
generalization error: 0.448850629320299
```

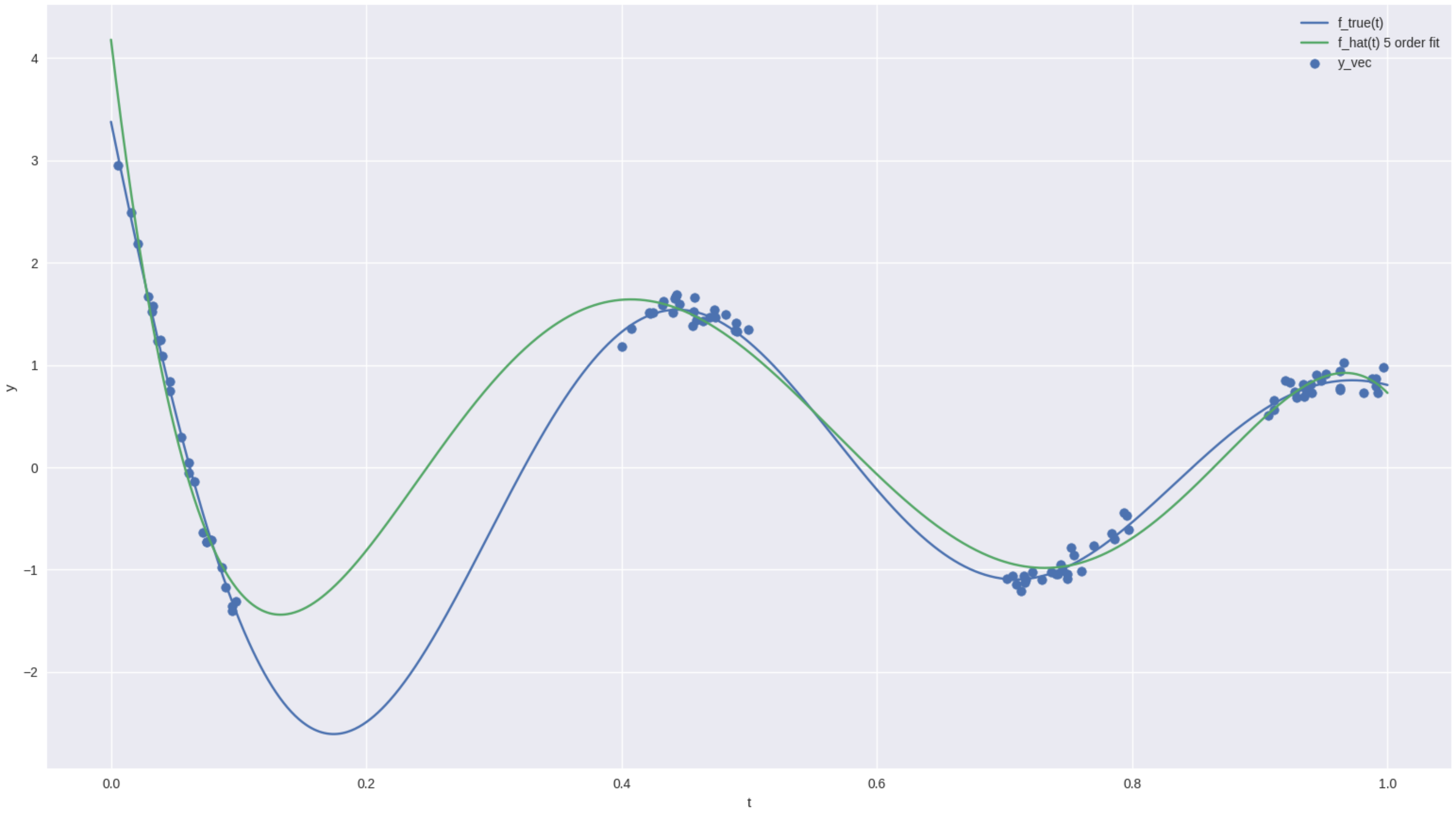
As we sweep the value of delta the sample error is pretty stable until it increases at a certain inflection point. The generalization error on the other hand, starts high and then decreases before continuing to climb again.


f_hat
f_true(t)
f_hat(t) 3 order fit
y_vec
y
t
3
2
1
0
−1
−2
0.0
0.2
0.4
0.6
0.8
1.0

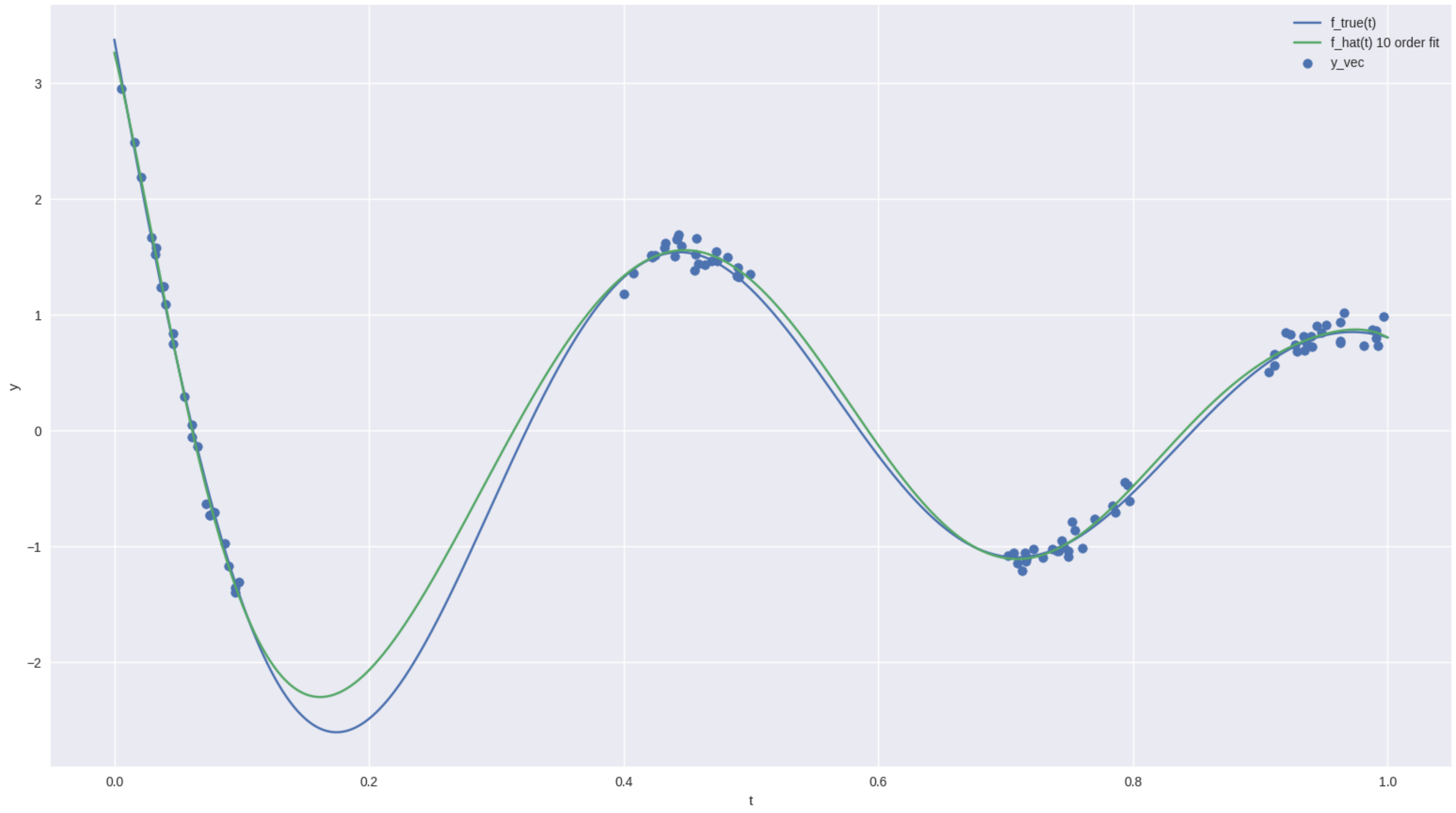

sample error
error
basis_num
1.4
1.2
1.0
0.8
5.0
7.5
10.0
12.5
15.0
17.5
20.0
22.5
25.0

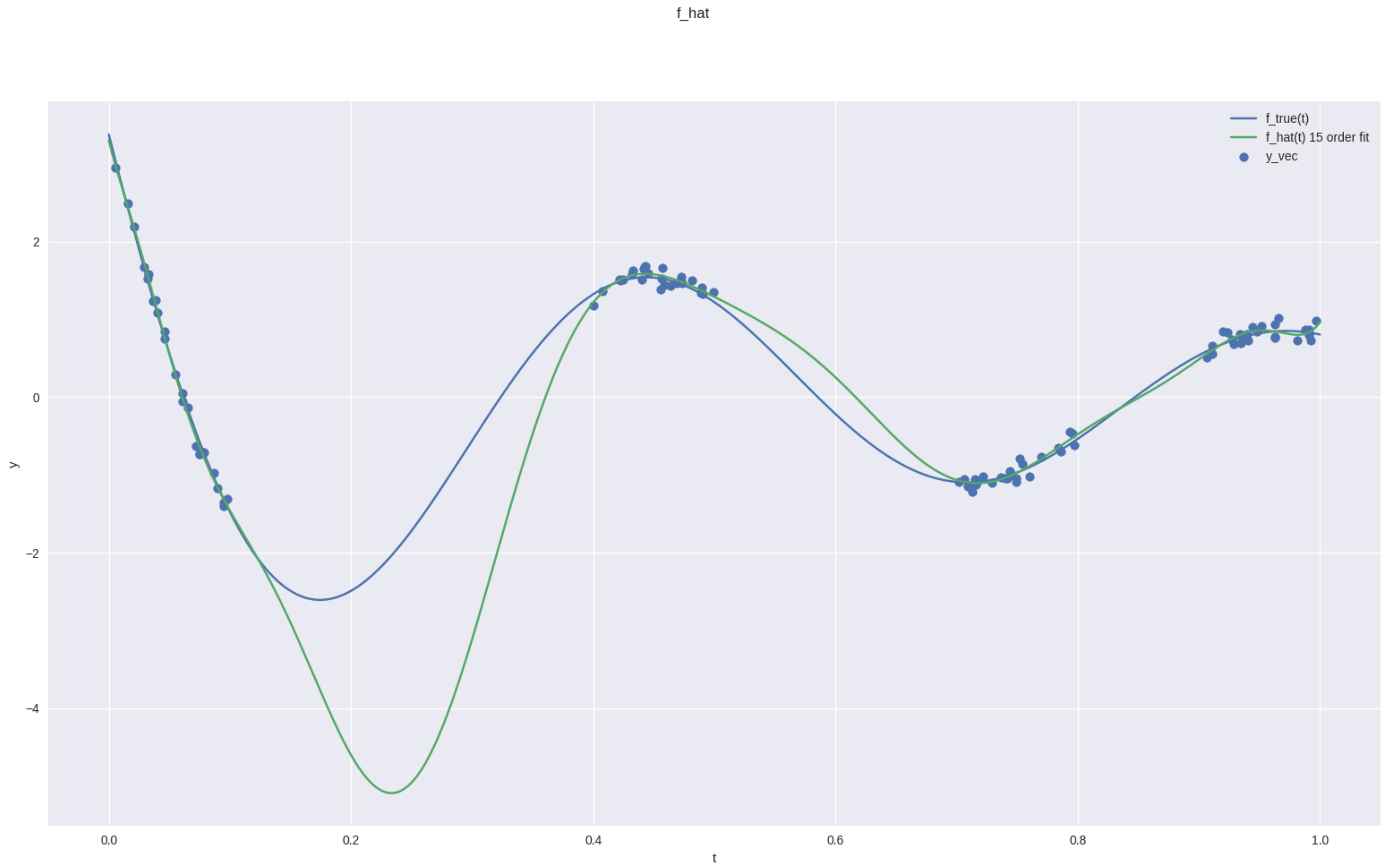

generalization error
error
basis_num
0
5
10
15
20
25
30
35
40
5.0
7.5
10.0
12.5
15.0
17.5
20.0
22.5
25.0

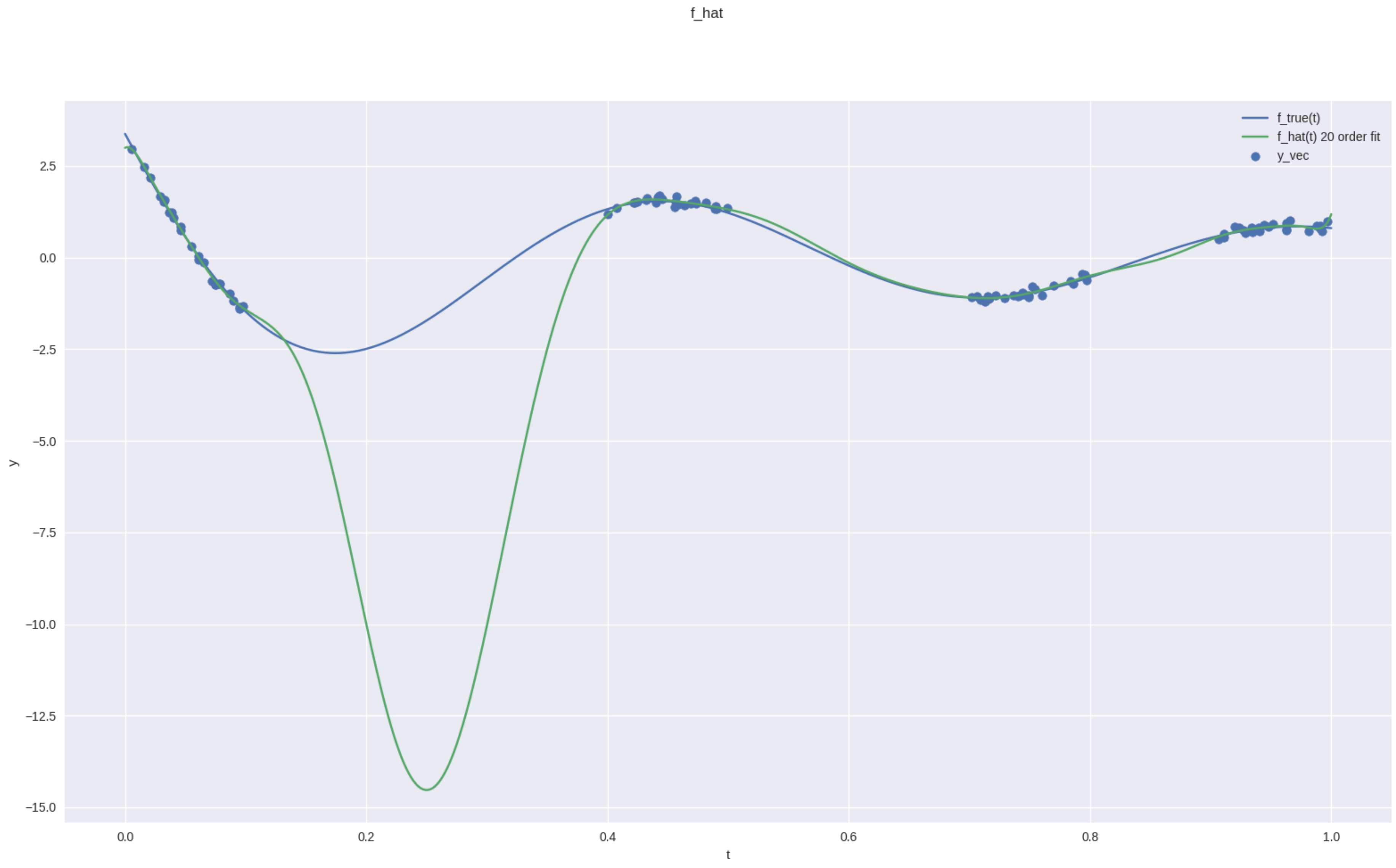
f_hat



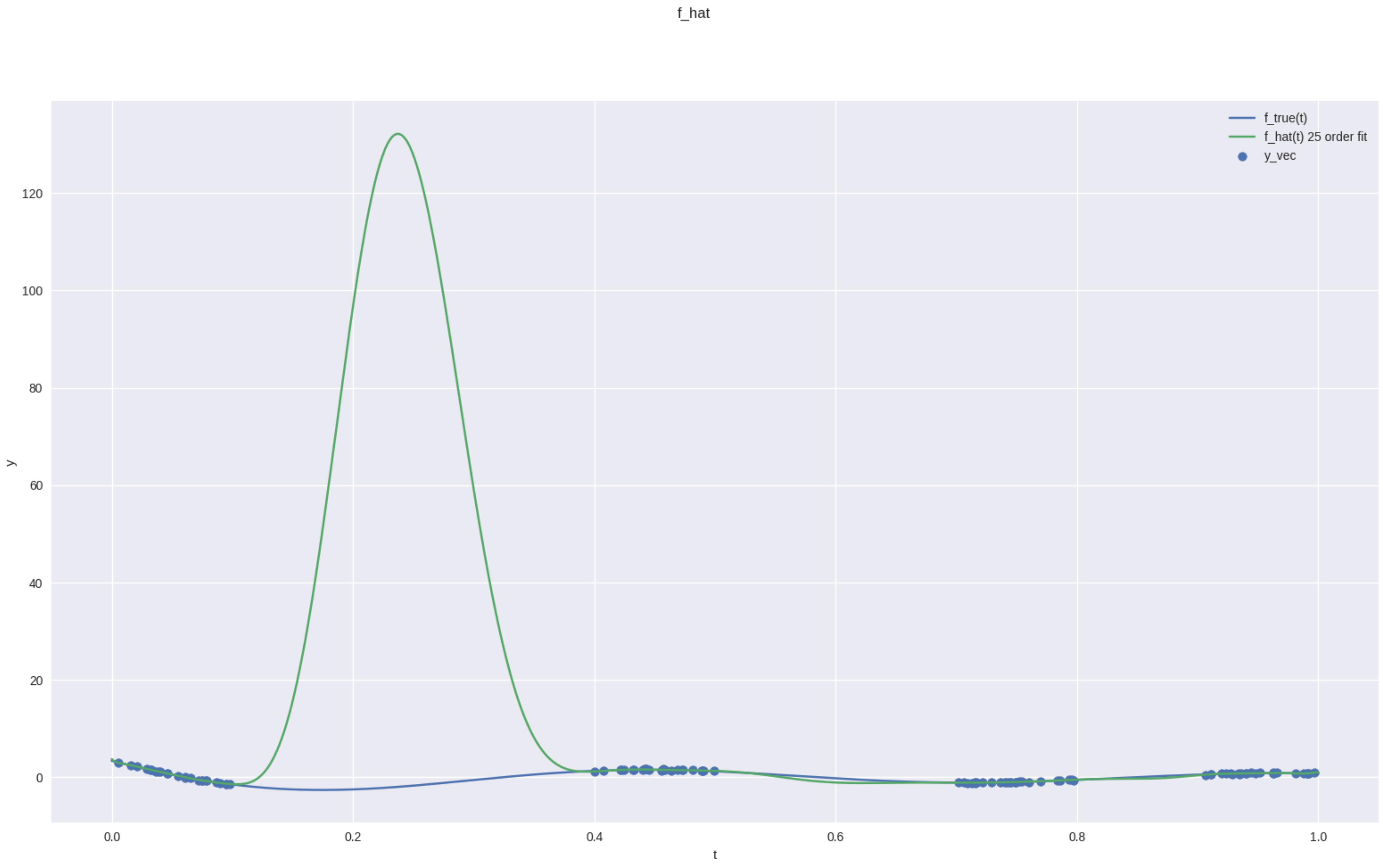
f_hat

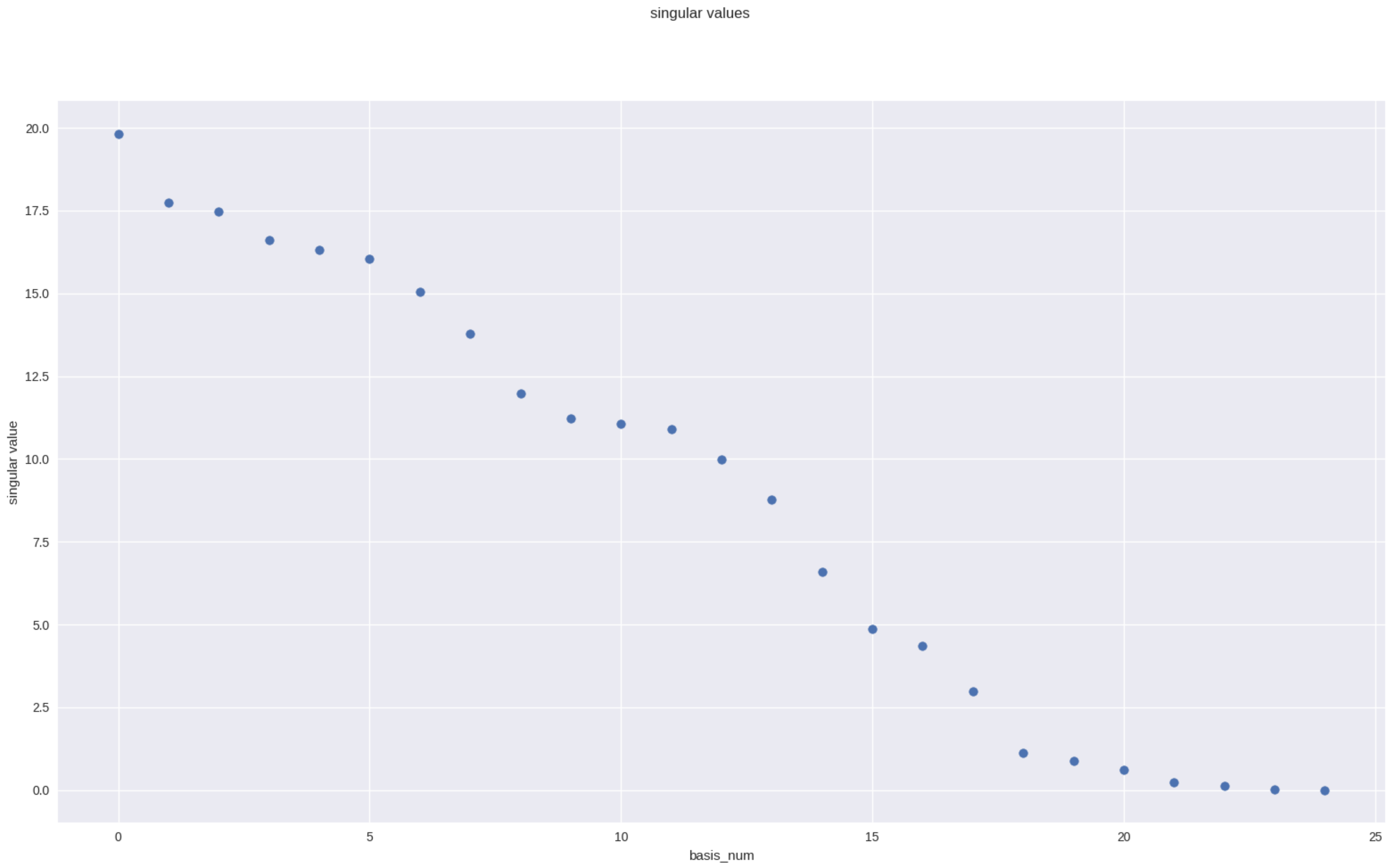



f_hat
f_true(t)
f_hat(t) 15 order fit
y_vec
y
t
2
0
−2
−4
0.0
0.2
0.4
0.6
0.8
1.0

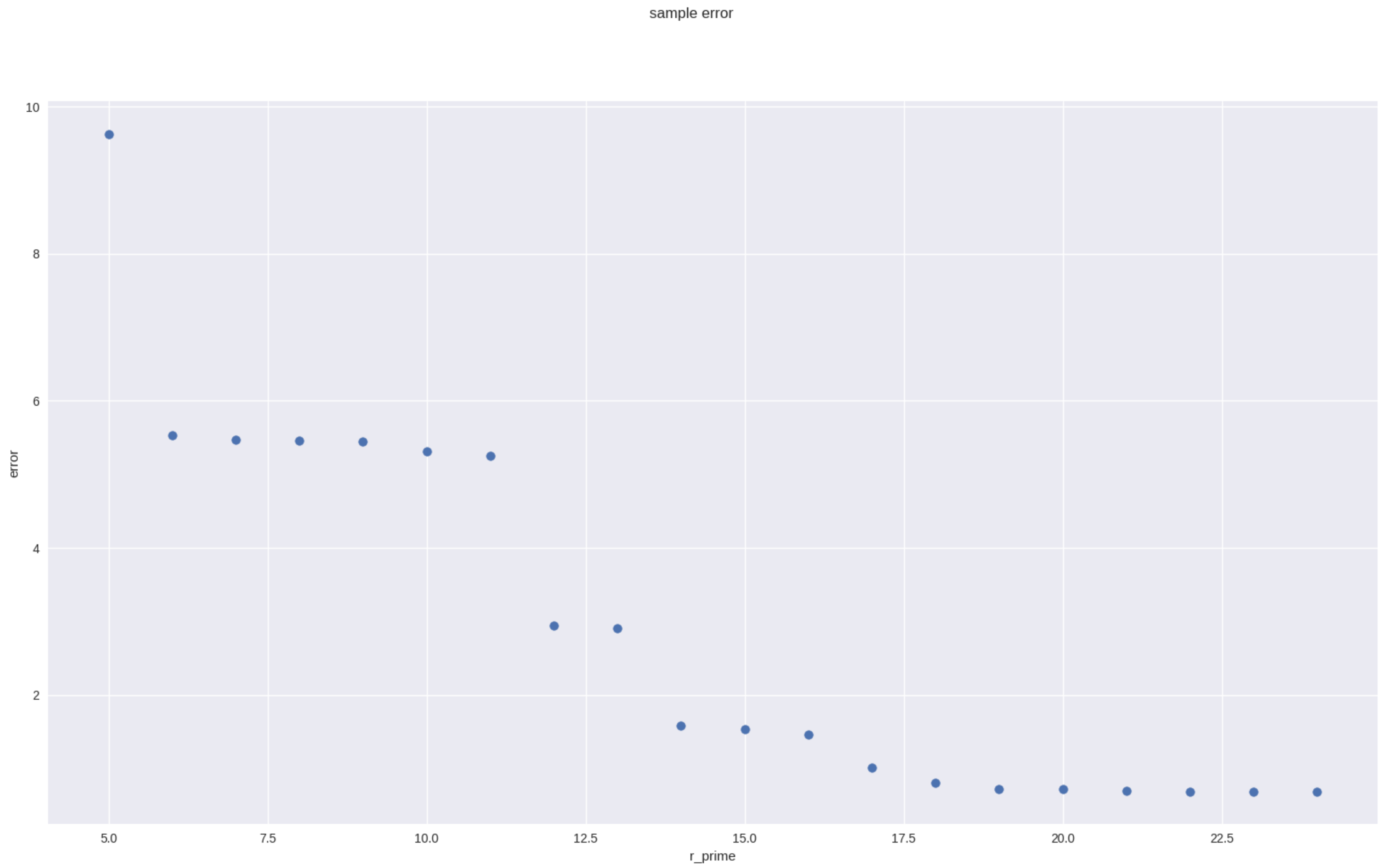


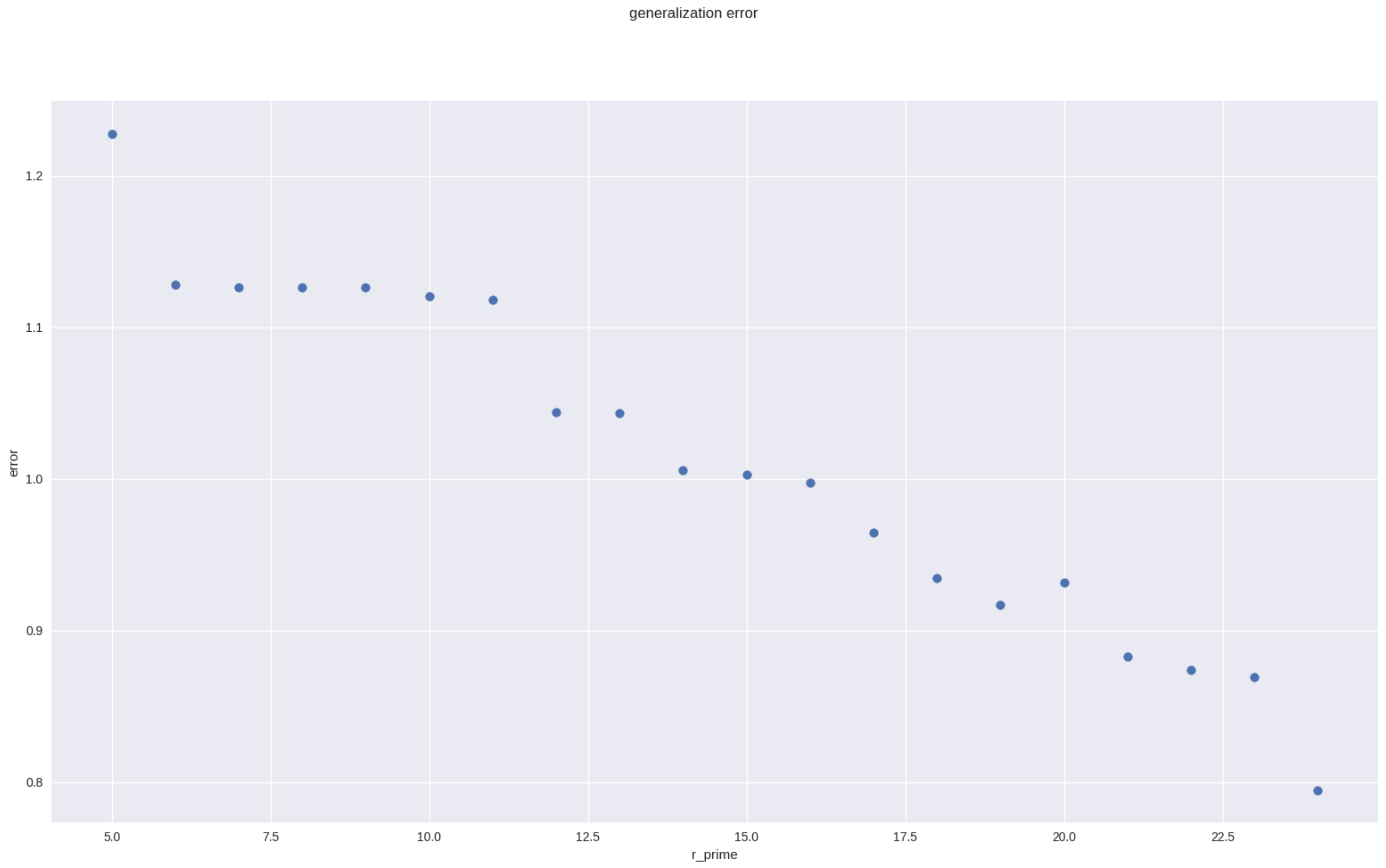
f_hat
f_true(t)
f_hat(t) 20 order fit
y_vec
2.5
0.0
−2.5
−5.0
−7.5
−10.0
−12.5
−15.0
y
0.0
0.2
0.4
0.6
0.8
1.0
t

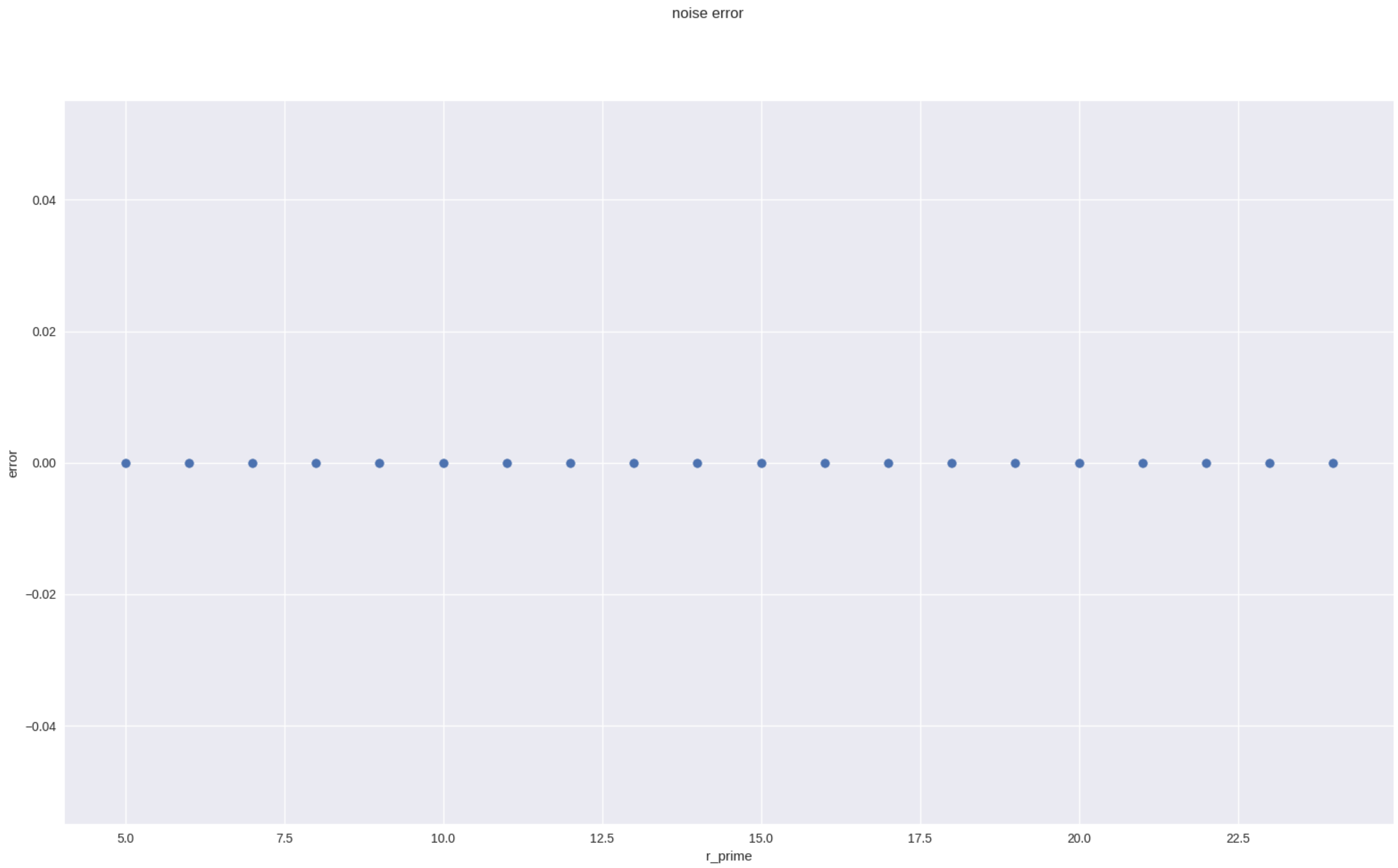

f_hat
f_true(t)
f_hat(t) 25 order fit
y_vec
y
t
0
20
40
60
80
100
120
0.0
0.2
0.4
0.6
0.8
1.0



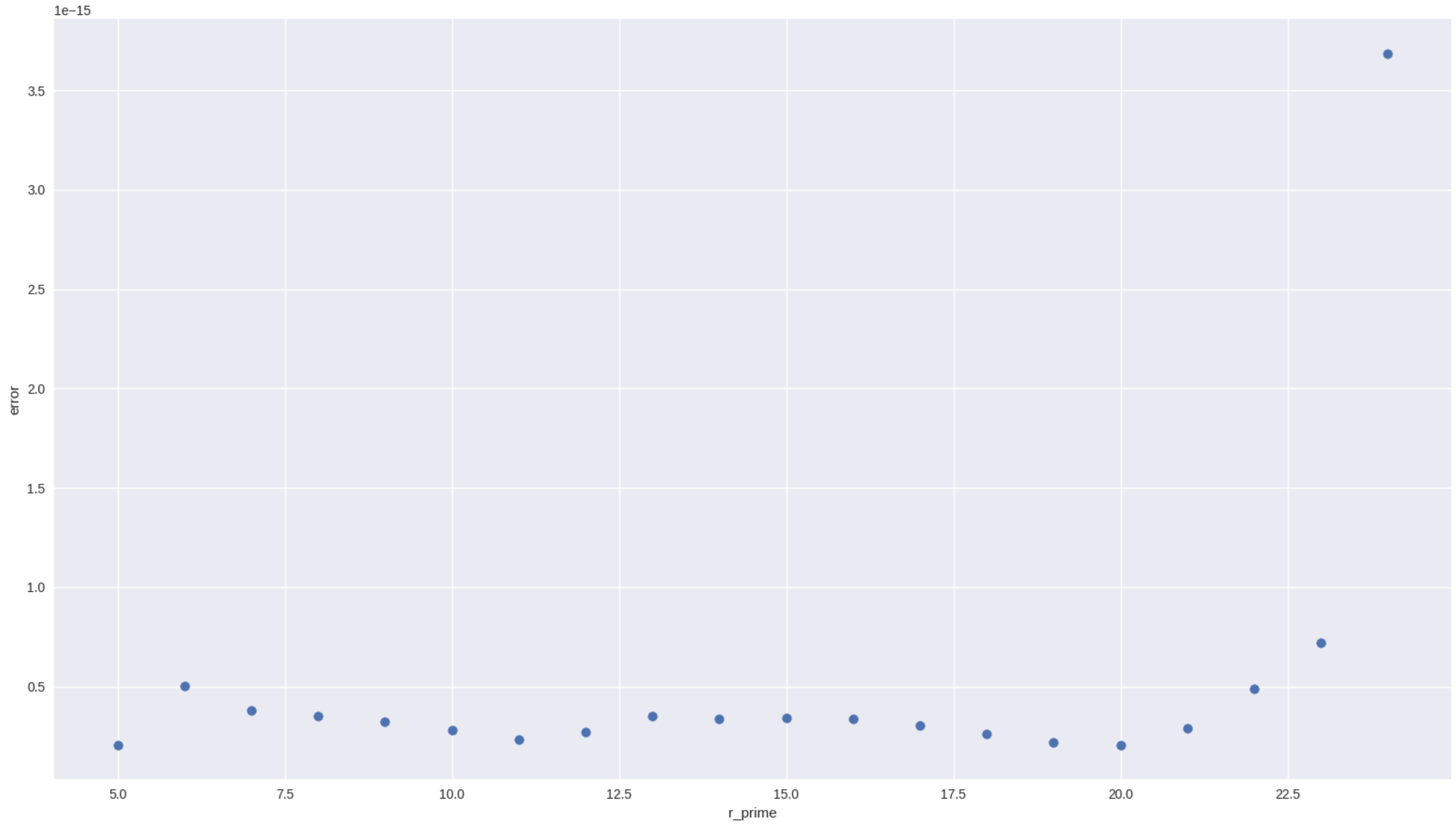
singular values
singular value
basis_num
20.0
17.5
15.0
12.5
10.0
7.5
5.0
2.5
0.0
0
5
10
15
20
25

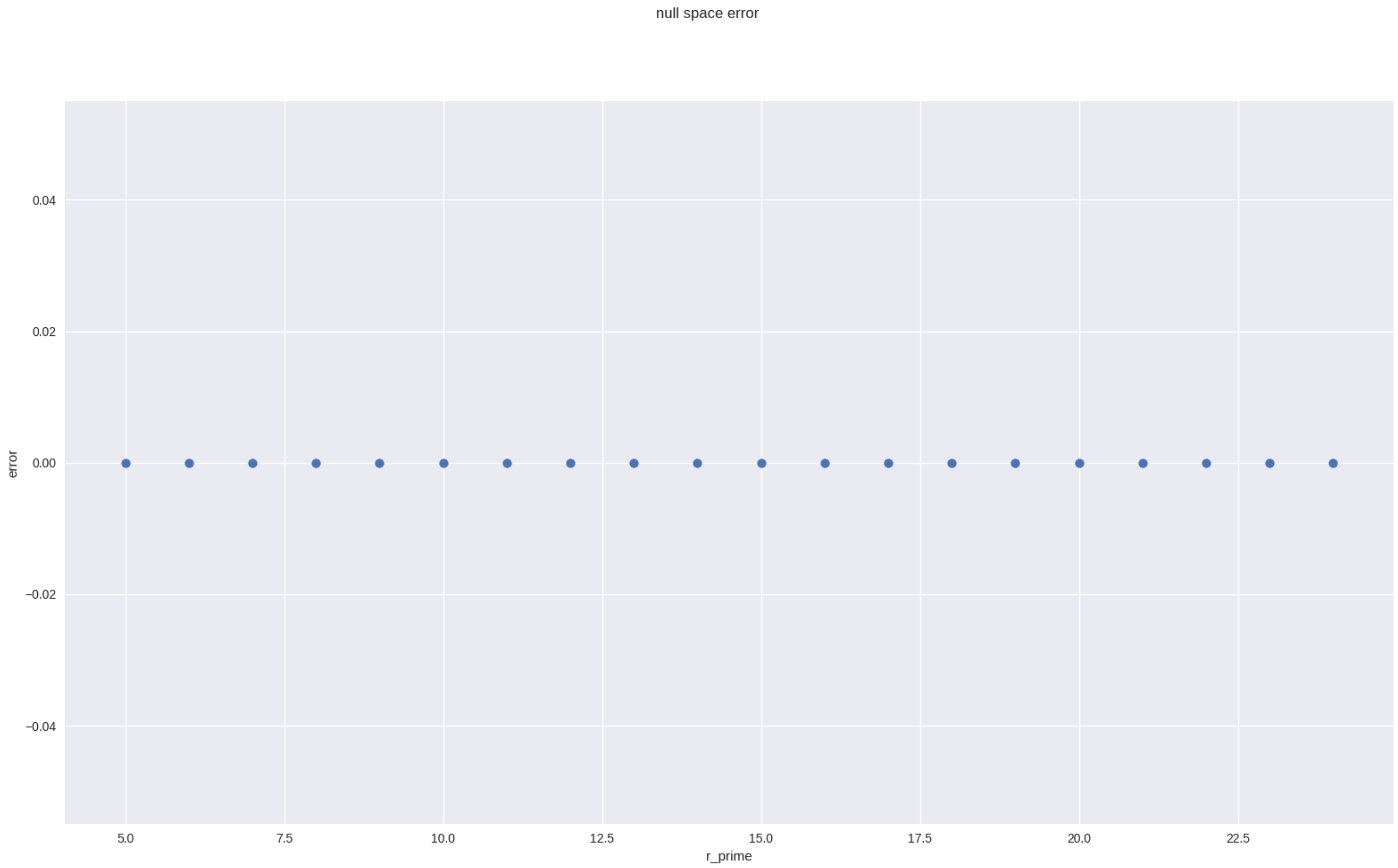

sample error
error
r_prime
10
8
6
4
2
5.0
7.5
10.0
12.5
15.0
17.5
20.0
22.5


generalization error
error
r_prime
1.2
1.1
1.0
0.9
0.8
5.0
7.5
10.0
12.5
15.0
17.5
20.0
22.5

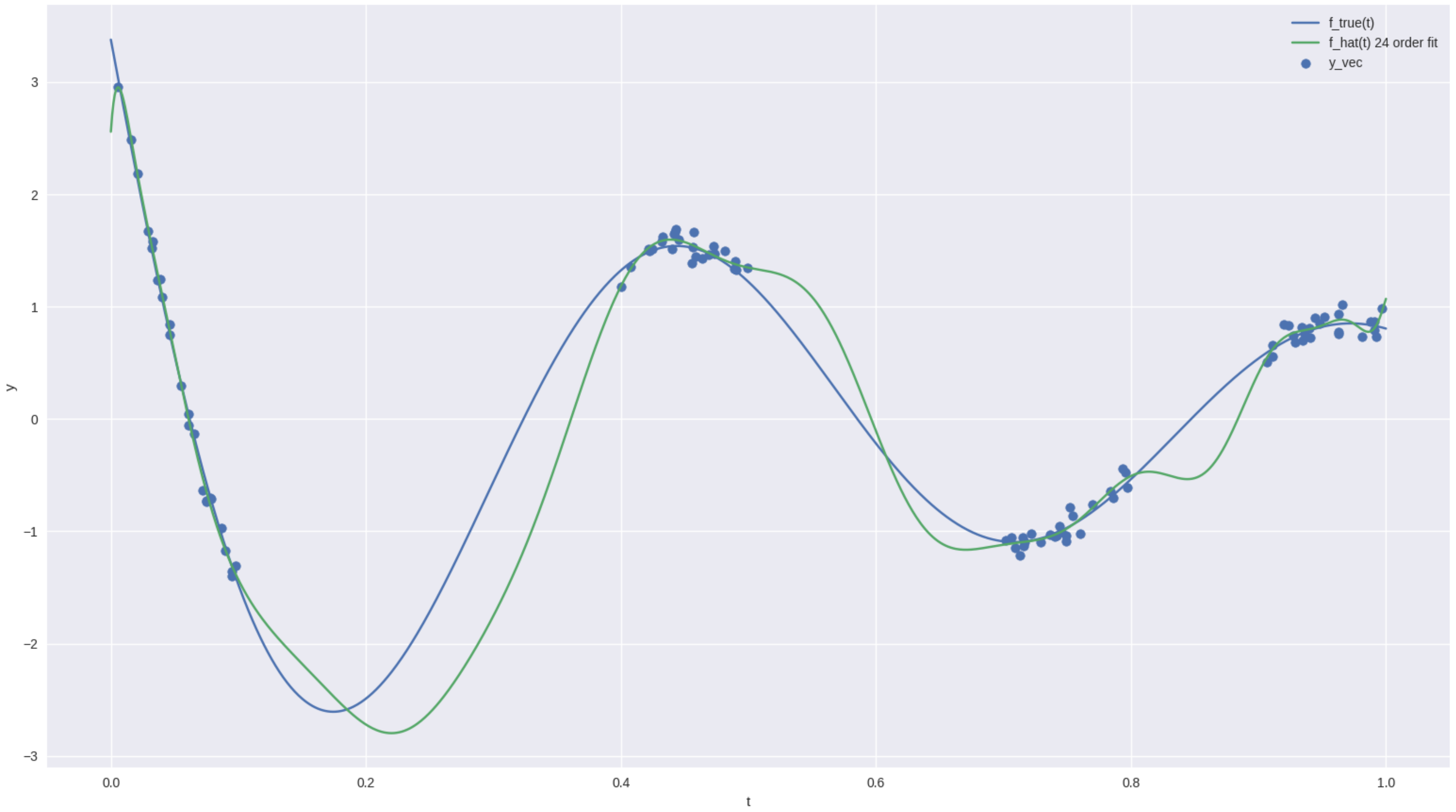

noise error
error
r_prime
0.04
0.02
0.00
−0.02
−0.04
5.0
7.5
10.0
12.5
15.0
17.5
20.0
22.5

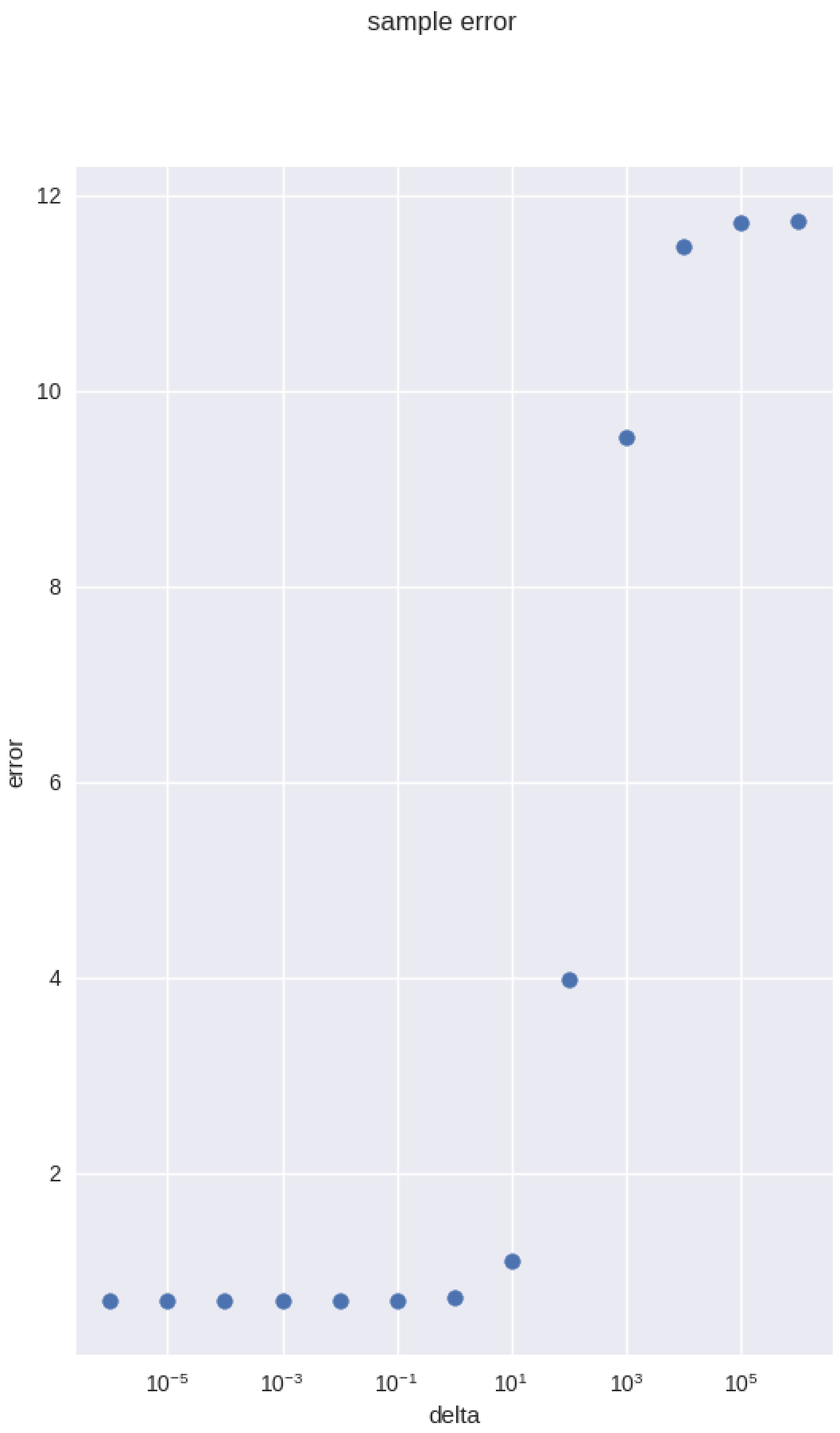
approx error

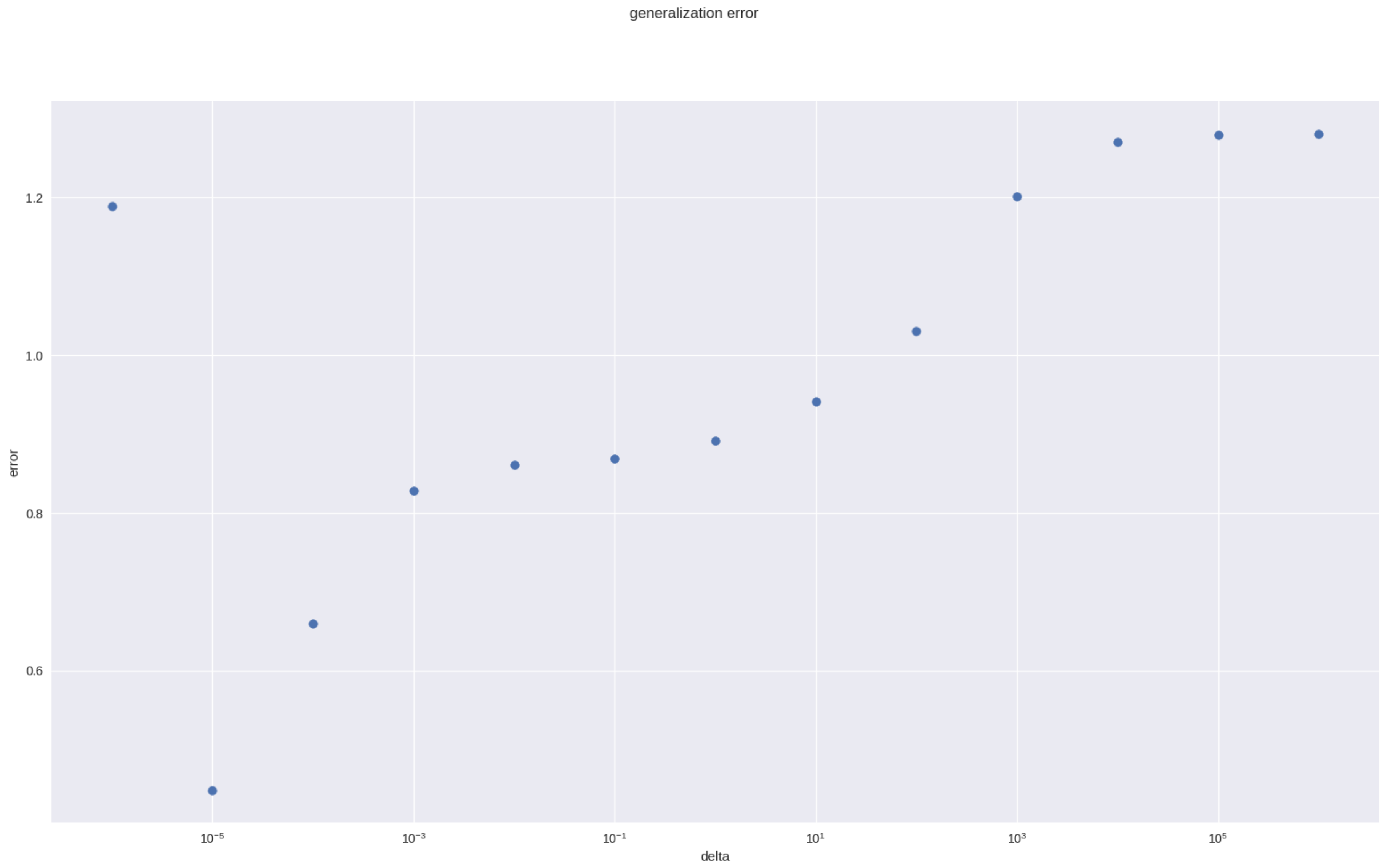



null space error
error
r_prime
0.04
0.02
0.00
−0.02
−0.04
5.0
7.5
10.0
12.5
15.0
17.5
20.0
22.5

f_hat delta=1e-05




sample error
error
12
10
8
6
4
2
10^{-5}
10^{-3}
10^{-1}
10^{1}
10^{3}
10^{5}
delta



generalization error
error
delta
1.2
1.0
0.8
0.6
10^{-5}
10^{-3}
10^{-1}
10^{1}
10^{3}
10^{5}

4a) Let A be a $N \times N$ symmetric matrix. Show that

$$\text{trace}(A) = \sum_{n=1}^{N} \lambda_n$$

where $\{\lambda_n\}$ are the eigenvalues of A.

$$\text{trace}(A) = \text{trace}(V \Lambda V^T) = \text{trace}(\Lambda V^T V) = \text{trace}(\Lambda)$$

$$= \sum_{n=1}^{N} \Lambda[n,n] = \sum_{n=1}^{N} \lambda_n \quad \checkmark$$

circular trace theorem

b) Now let A be an arbitrary $M \times N$ matrix. Recall the definition of the Frobenius norm:

$$\|A\|_F = \left(\sum_{m=1}^{M} \sum_{n=1}^{N} |A[m,n]|^2 \right)^{1/2}$$

Show that

$$\|A\|_F^2 = \text{trace}(A^T A) = \sum_{r=1}^{R} \sigma_r^2$$

where R is the rank of A and the $\{\sigma_r\}$ are the singular values of A.

$$\|A_F\|^2 = \sum_{m=1}^{M} \sum_{n=1}^{N} |A[m,n]|^2 = \sum_{n=1}^{N} \left(\sum_{m=1}^{M} |A[m,n]|^2 \right) = \sum_{n=1}^{N} A^T A[n,n] = \text{trace}(A^T A)$$

$$\text{trace}(A^T A) = \text{trace}(V \Sigma^T U^T U \Sigma V^T)$$

$$= \text{trace}(V \Sigma^T \Sigma V^T) = \text{trace}(\Sigma^T \Sigma V^T V)$$

$M \times M \cdot M \times N \cdot N \times N$

$N \times M \cdot M \times N$

$$= \text{trace}(\Sigma^T \Sigma) = \sum_{n=1}^{N} \Sigma^T \Sigma[n,n] = \sum_{n=1}^{N} \sigma_n^2 = \sum_{r=1}^{R} \sigma_r^2$$

since entries $r > N$ are zero.

c) The operator norm (sometimes called the spectral norm) of an $M \times N$ matrix is $\|A\| = \max_{x \in \mathbb{R}^N, \|x\|_2 = 1} \|Ax\|_2$

also as "the" matrix norm

Show that $\|A\| = \sigma_1$,

where σ_1 is the largest singular value of A.

$$\|Ax\|_2 = \|U \Sigma V^T x\|_2$$

$$\|Ax\|_2^2 = \|U \Sigma V^T x\|_2^2 = \langle U \Sigma V^T x, U \Sigma V^T x \rangle$$

$$= \sum_{n=1}^{N} (u_n \sigma_n v_n^T x_n)^T (u_n \sigma_n v_n^T x_n)$$

$$= \sum_{n=1}^{N} x_n \sigma_n v_n \overset{1}{\cancel{u_n^T u_n}} v_n^T \sigma_n x_n$$

$$= \sum_{n=1}^{N} x_n \sigma_n \overset{1}{\cancel{v_n v_n^T}} \sigma_n x_n$$

$$= \sum_{n=1}^{N} \sigma_n^2 x_n^2 = \sum_{n=1}^{N} (x_n \sigma_n)^2 = \langle x_n, \sigma_n \rangle$$

$$\|Ax\|_2^2 = \langle x_n, \sigma_n \rangle$$

$\|Ax\|_2 = \sqrt{\langle x_n, \sigma_n \rangle} \leftarrow$ is maximized when x_1 of $x = 1$, gives all budget to largest singular value. Thus $\|A\| = \max \|Ax\|_2 = \sigma_1$

$\underset{\sigma_1}{\uparrow}$

For which x does

$\|Ax\|_2 = \|A\| \cdot \|x\|_2$?

From the above proof: x where $x_1 = 1$ and the rest are all zeros gives us the equality.

d) Prove that $\|A\| \leq \|A\|_F$. Give an example of an A with $\|A\| = \|A\|_F$.

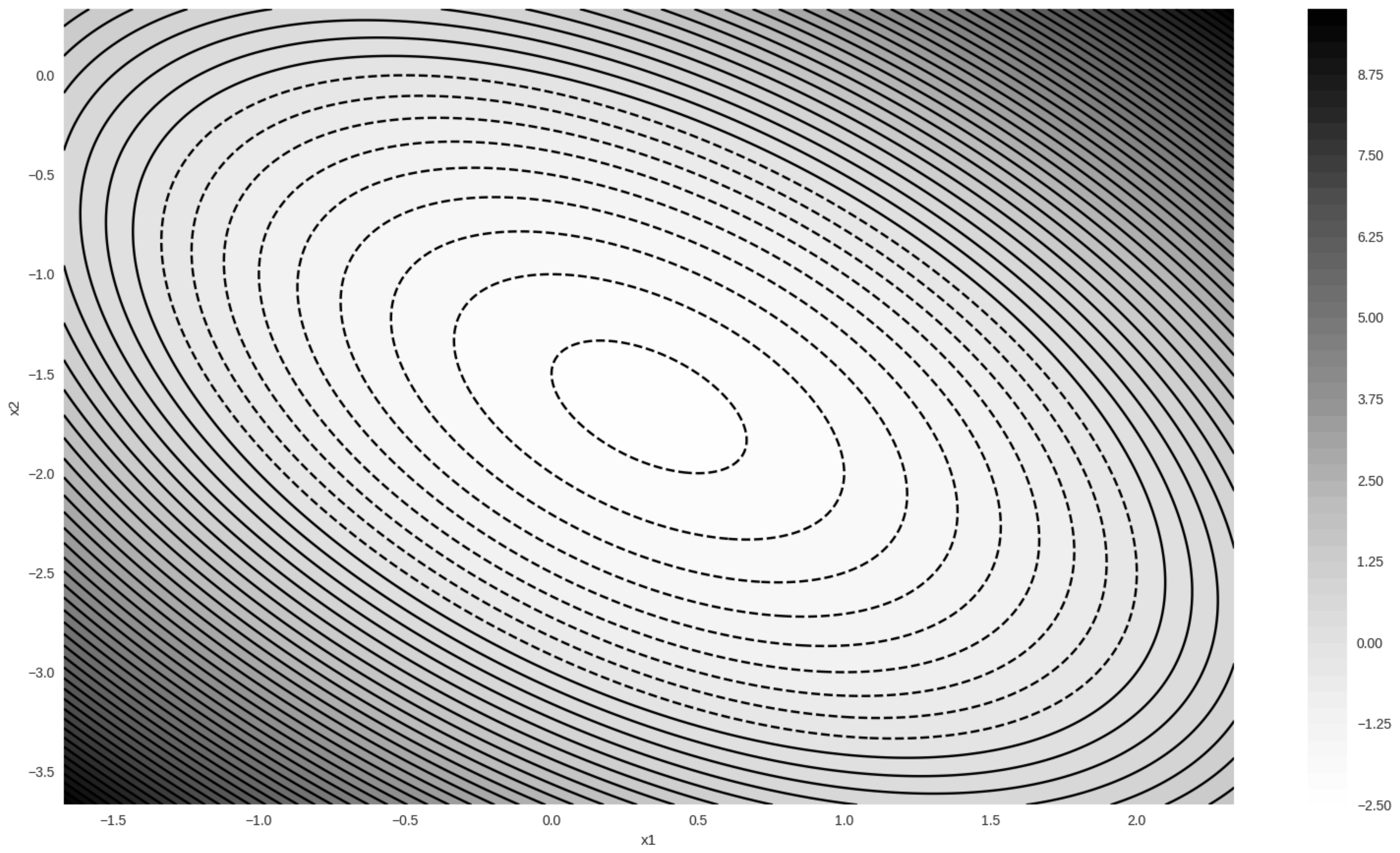
$$\|A\|^2 = \sigma_1^2 \leq \sum_{r=1}^{R} \sigma_r^2 = \|A\|_F^2$$

$\Rightarrow \|A\|^2 \leq \|A\|_F^2 \Rightarrow \|A\| \leq \|A\|_F$ ✓

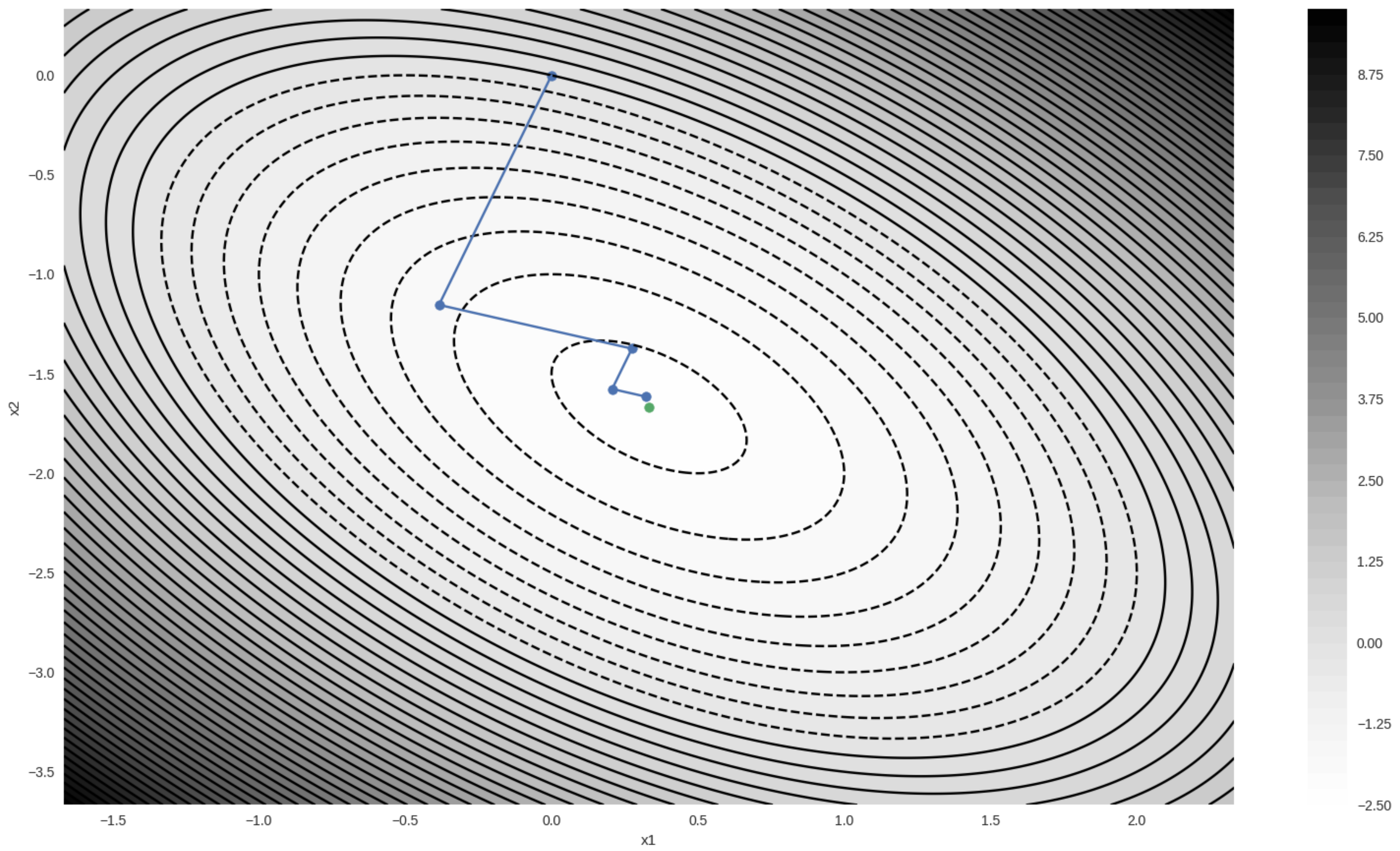
For an A rank 1, $\sum_{r=1}^{R} \sigma_r^2$ will equal σ_1, thus $\|A\|^2 = \|A\|_F^2$ and $\|A\| = \|A\|_F$ ✓

```python
"""Problem 5."""

import matplotlib as mpl
import matplotlib.pyplot as plt

import numpy as np


mpl.style.use('seaborn')

h_mat_problem = np.array([[2, 1], [1, 2]])
b_vec_problem = np.array([-1, -3])


def part_a(h_mat, b_vec):
    """Part a."""
    print('Part a')
    # f_grad = Hx - b = 0
    f_argmin = np.linalg.inv(h_mat) @ b_vec
    f_min = 1/2 * np.transpose(f_argmin) @ h_mat @ f_argmin - \
        np.transpose(b_vec) @ f_argmin

    print('f_min: ' + str(f_min))
    print('f_argmin: ' + str(f_argmin))

    return f_argmin


f_argmin_part_a = part_a(h_mat_problem, b_vec_problem)


def part_b(h_mat, b_vec):
    """Part b."""
    print('Part b')
    x1_sqrd_coeff = 1/2 * h_mat[0, 0]
    x2_sqrd_coeff = 1/2 * h_mat[1, 1]
    x1x2_coeff = 1/2 * (h_mat[0, 1] + h_mat[1, 0])
    x1_coeff = -1 * b_vec[0]
    x2_coeff = -1 * b_vec[1]
    f_coeff = np.array([x1_sqrd_coeff, x2_sqrd_coeff, x1x2_coeff, x1_coeff,
                        x2_coeff])
    print('f_coeff: ' + str(f_coeff))
    return f_coeff


f_coeff_part_b = part_b(h_mat_problem, b_vec_problem)


def plot_contour(f_argmin, f_coeff):
    """Plot Contour."""
    fig = plt.figure()
    fig.suptitle('Contour plot of f(x)')
    axis = fig.add_subplot(111)

    interval = 2

    x1_vec = np.linspace(f_argmin[0] - interval,
                         f_argmin[0] + interval, num=1000)
    x2_vec = np.linspace(f_argmin[1] - interval,
                         f_argmin[1] + interval, num=1000)

    x1_mat, x2_mat = np.meshgrid(x1_vec, x2_vec)

    f_mat = f_coeff[0] * x1_mat ** 2 + f_coeff[1] * x2_mat ** 2 + \
        f_coeff[2] * x1_mat * x2_mat + \
        f_coeff[3] * x1_mat + f_coeff[4] * x2_mat

    csetf = axis.contourf(x1_mat, x2_mat, f_mat, levels=50)
    axis.contour(x1_mat, x2_mat, f_mat, csetf.levels, colors='k')

    fig.colorbar(csetf, ax=axis)

    axis.set_xlabel('x1')
    axis.set_ylabel('x2')

    return axis


def part_c(f_argmin, f_coeff, h_mat):
    """Part c."""
    print('Part c')

    # Contour plot of f(x)
    plot_contour(f_argmin, f_coeff)
    plt.show()

    # Compute eigenvectors and eigenvalues of h_mat
    eigenvalue_vec, eigenvector_mat = np.linalg.eig(h_mat)
    print('Eigenvalues: \n' + str(eigenvalue_vec))
    print('Eigenvectors: \n' + str(eigenvector_mat))

    print("""
    The eigenvectors are providing the direction of the eclipse structure we see
    and the eigenvalues can be related to the magnitude of the major/minor axis
    of the contour of our function.
    """)


part_c(f_argmin_part_a, f_coeff_part_b, h_mat_problem)


def gdstep(h_mat, x_vec, r_vec):
    """Gradient Descent Step."""
    q_vec = h_mat @ r_vec
    alpha = (np.transpose(r_vec) @ r_vec) / (np.transpose(r_vec) @ q_vec)
    x_vec = x_vec + alpha * r_vec
    r_vec = r_vec - alpha * q_vec

    return [x_vec, r_vec]


def part_d(h_mat, b_vec, f_argmin, f_coeff):
    """Part d."""
    print('Part d')

    maxiter = 4

    x_vec_list = [None] * (maxiter + 1)
    x_vec_list[0] = np.zeros(2)

    r_vec_list = [None] * (maxiter + 1)
    r_vec_list[0] = b_vec - h_mat @ x_vec_list[0]

    for index in range(1, maxiter + 1):
        x_vec_list[index], r_vec_list[index] = \
            gdstep(h_mat, x_vec_list[index-1], r_vec_list[index-1])

    # pylint: disable=unsubscriptable-object  # pylint/issues/3139
    x1_vec = [x_vec[0] for x_vec in x_vec_list]
    x2_vec = [x_vec[1] for x_vec in x_vec_list]

    axis = plot_contour(f_argmin, f_coeff)
    axis.scatter(x1_vec, x2_vec)
    axis.plot(x1_vec, x2_vec)
    axis.scatter(f_argmin[0], f_argmin[1])

    plt.show()


part_d(h_mat_problem, b_vec_problem, f_argmin_part_a, f_coeff_part_b)


def cgstep(h_mat, x_vec, r_vec, d_vec):
    """Conjugate Gradient Step."""
    alpha = (np.transpose(r_vec) @ r_vec) / \
        (np.transpose(d_vec) @ h_mat @ d_vec)
    x_vec = x_vec + alpha * d_vec
    r_vec_tmp = r_vec
    r_vec = r_vec - alpha * h_mat @ d_vec
    beta = (np.transpose(r_vec) @ r_vec) / \
        (np.transpose(r_vec_tmp) @ r_vec_tmp)
    d_vec = r_vec + beta * d_vec

    return [x_vec, r_vec, d_vec]


def part_e(h_mat, b_vec, f_argmin, f_coeff):
    """Part e."""
    print('Part e')

    maxiter = 4

    x_vec_list = [None] * (maxiter + 1)
    x_vec_list[0] = np.zeros(2)

    r_vec_list = [None] * (maxiter + 1)
    r_vec_list[0] = b_vec - h_mat @ x_vec_list[0]

    d_vec_list = [None] * (maxiter + 1)
    d_vec_list[0] = b_vec

    for index in range(1, maxiter + 1):
        x_vec_list[index], r_vec_list[index], d_vec_list[index] = \
            cgstep(h_mat, x_vec_list[index-1], r_vec_list[index-1],
                   d_vec_list[index-1])

    # pylint: disable=unsubscriptable-object  # pylint/issues/3139
    x1_vec = [x_vec[0] for x_vec in x_vec_list]
    x2_vec = [x_vec[1] for x_vec in x_vec_list]

    axis = plot_contour(f_argmin, f_coeff)
    axis.scatter(x1_vec, x2_vec)
    axis.plot(x1_vec, x2_vec)
    axis.scatter(f_argmin[0], f_argmin[1])

    plt.show()


part_e(h_mat_problem, b_vec_problem, f_argmin_part_a, f_coeff_part_b)
```

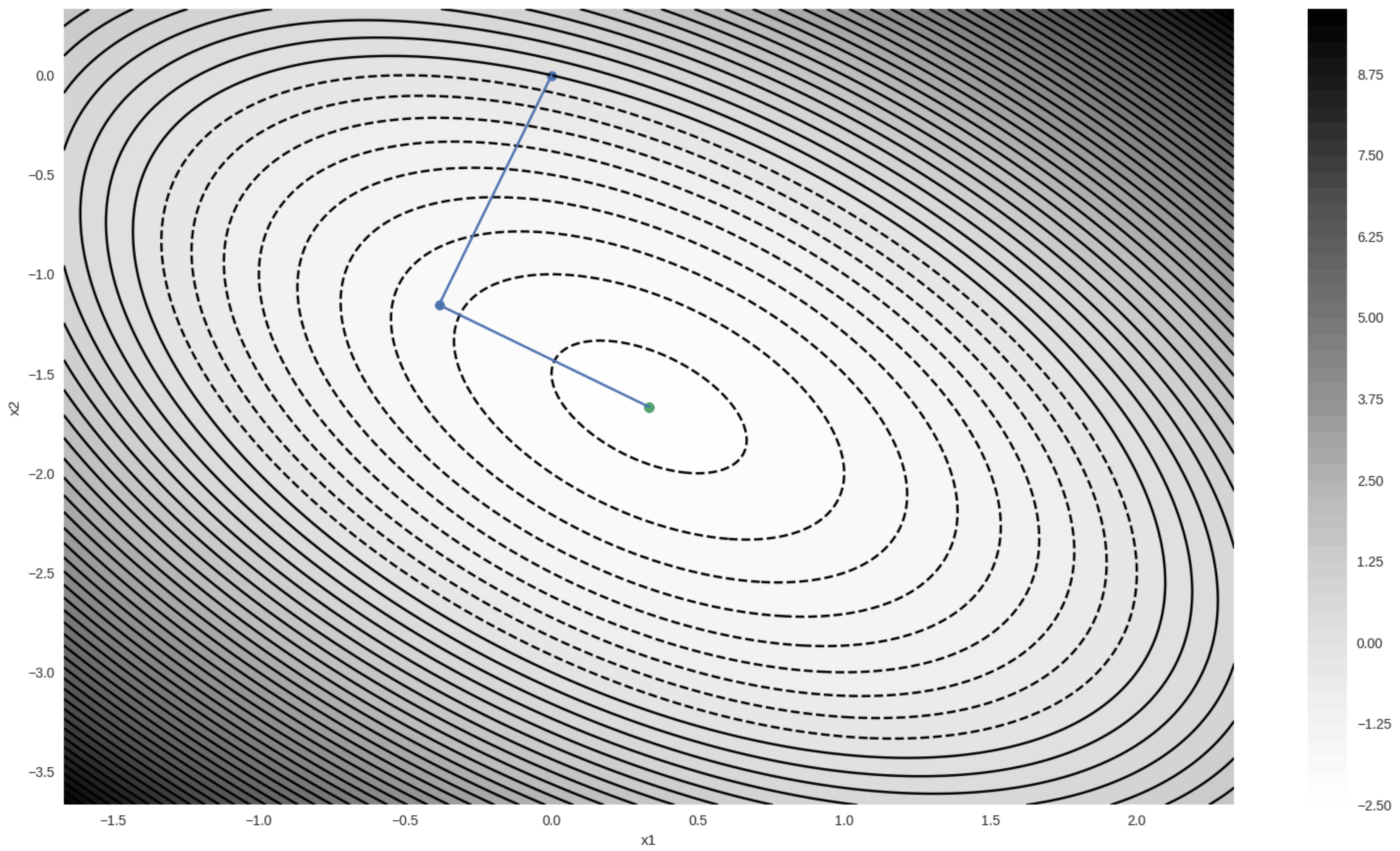
Contour plot of f(x)



Contour plot of f(x)



Contour plot of f(x)



```
Part a
f_min: -2.3333333333333335
f_argmin: [ 0.33333333 -1.66666667]
Part b
f_coeff: [1. 1. 1. 1. 3.]
Part c
Eigenvalues:
[3. 1.]
Eigenvectors:
[[ 0.70710678 -0.70710678]
 [ 0.70710678  0.70710678]]

    The eigenvectors are providing the direction of the eclipse structure we see
    and the eigenvalues can be related to the magnitude of the major/minor axis
    of the contour of our function.

Part d
Part e
```

```python
"""Problem 6."""

import numpy as np

import scipy.io as sio

MAT_FILENAME = 'hw06p6_data.mat'
data_samples = sio.loadmat(MAT_FILENAME)

H_data = data_samples['H']
b_data = data_samples['b']


def gdstep(h_mat, x_vec, r_vec):
    """Gradient Descent Step."""
    q_vec = h_mat @ r_vec
    alpha = (np.transpose(r_vec) @ r_vec) / (np.transpose(r_vec) @ q_vec)
    x_vec = x_vec + alpha * r_vec
    r_vec = r_vec - alpha * q_vec

    return [x_vec, r_vec]


def gdsolve(h_mat, b_vec, tol, maxiter):
    """Gradient Descent."""
    _iter = 0
    x_vec = np.zeros((np.size(b_vec)))
    r_vec = b_vec - h_mat @ x_vec

    while np.linalg.norm(r_vec)/np.linalg.norm(b_vec) >= tol \
            and _iter < maxiter:
        x_vec, r_vec = gdstep(h_mat, x_vec, r_vec)
        _iter = _iter + 1

    return [x_vec, _iter]


def part_a(h_mat, b_vec):
    """Part a."""
    print('Part a')

    tol = 1e-6
    maxiter = np.inf

    x_hat, _iter = gdsolve(h_mat, b_vec, tol, maxiter)
    print('Iterations: ' + str(_iter))

    error = np.linalg.norm(h_mat @ x_hat - b_vec)
    print('Error: ' + str(error))


part_a(H_data, b_data)


def cgstep(h_mat, x_vec, r_vec, d_vec):
    """Conjugate Gradient Step."""
    alpha = (np.transpose(r_vec) @ r_vec) / \
        (np.transpose(d_vec) @ h_mat @ d_vec)
    x_vec = x_vec + alpha * d_vec
    r_vec_tmp = r_vec
    r_vec = r_vec - alpha * h_mat @ d_vec
    beta = (np.transpose(r_vec) @ r_vec) / \
        (np.transpose(r_vec_tmp) @ r_vec_tmp)
    d_vec = r_vec + beta * d_vec
    return [x_vec, r_vec, d_vec]


def cgsolve(h_mat, b_vec, tol, maxiter):
    """Conjugate Gradient."""
    _iter = 0
    x_vec = np.zeros((np.size(b_vec)))
    r_vec = b_vec - h_mat @ x_vec
    d_vec = r_vec

    while np.linalg.norm(r_vec)/np.linalg.norm(b_vec) >= tol \
            and _iter < maxiter:

        x_vec, r_vec, d_vec = cgstep(h_mat, x_vec, r_vec, d_vec)

        _iter = _iter + 1

    return [x_vec, _iter]


def part_b(h_mat, b_vec):
    """Part b."""
    print('Part b')

    tol = 1e-6
    maxiter = np.inf

    x_hat, _iter = cgsolve(h_mat, b_vec, tol, maxiter)
    print('Iterations: ' + str(_iter))

    error = np.linalg.norm(h_mat @ x_hat - b_vec)
    print('Error: ' + str(error))

    print("""
Conjugate gradient converges in noticeably fewer iterations.
    """)


part_b(H_data, b_data)
```

```
Part a
Iterations: 230
Error: 2.8965427927627244e-05
Part b
Iterations: 49
Error: 2.799226186317398e-05
```

Conjugate gradient converges in noticeably fewer iterations.