SOD314 - Project

April 7, 2023

1 Introduction

Thomas Boyer, ENSTA Paris, 2023.

This notebook / PDF file is my report for the Numerical Project in Python "Cooperative Kernel regression" of SOD314 - Cooperative Optimization for data science.

How to run it: - I used Python `3.10.10`, but any reasonably close Python version should work. - Install the requirements either with `poetry install` or with `pip install -r requirements.txt`

The source code is in the `src` folder, with the main algorithms in `algs.py`.

The figures are drawn with the `plotly` library, and are thus interactive.

2 Imports

2.1 Local

[1]:
```
%load_ext autoreload
%autoreload 2
```

[2]:
```
from src.algs import run_ADMM, run_DD, run_DGD, run_FedAvg, run_GT
from src.utils import (
    build_A,
    build_kernel_matrices,
    build_kernel_matrices_out_of_dataset,
    check_W,
    plot_f,
    plot_opt_gap,
    plot_opt_gap_per_agent,
    print_vector_norms,
    show_selected_points,
    susbample_data,
    study_FedAvg,
)
```

2.2 External

[3]:
```
import pickle
from math import ceil, sqrt

import numpy as np
import plotly.express as px
import plotly.graph_objects as go
```

[4]:
```
import plotly.io as pio
pio.renderers.default = "notebook+pdf"
```

3 Reproducibility

Let's set the seed for reproducibility.

[5]:
```
rng = np.random.default_rng(42)
```

4 Load data

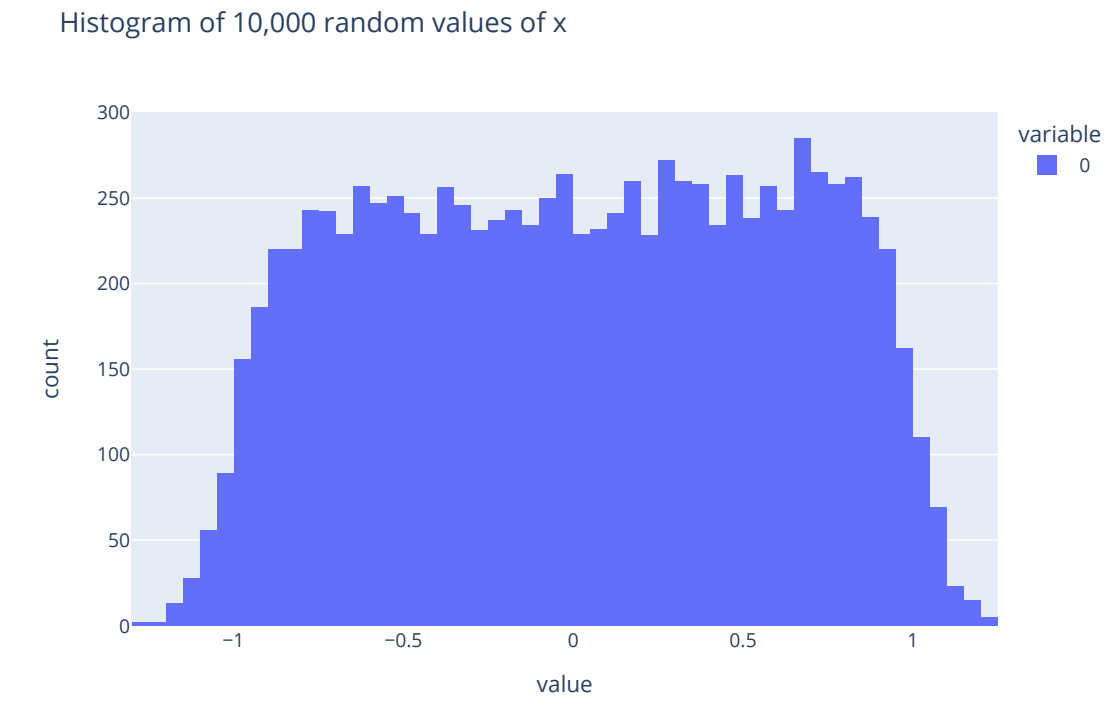
[6]:
```
with open("first_database.pkl", "rb") as f:
    x, y = pickle.load(f)

n_tot: int = x.size
assert n_tot == y.size and len(x.shape) == 1 and len(y.shape) == 1
```

5 Visualize data

I will plot only $10,000$ data points, randomly sampled for each plot.

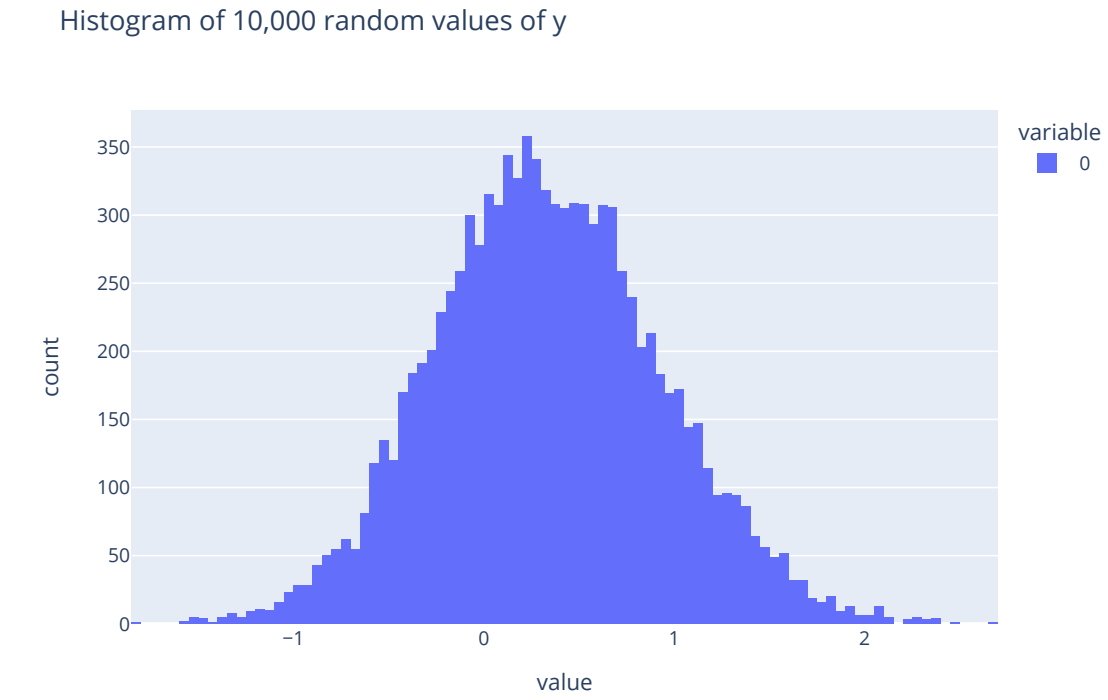
Let's start with some histograms:

[7]:
```
plot_data = rng.choice(x, size=10000, replace=False)
px.histogram(plot_data, title="Histogram of 10,000 random values of x")
```



Loading [MathJax]/extensions/MathMenu.js

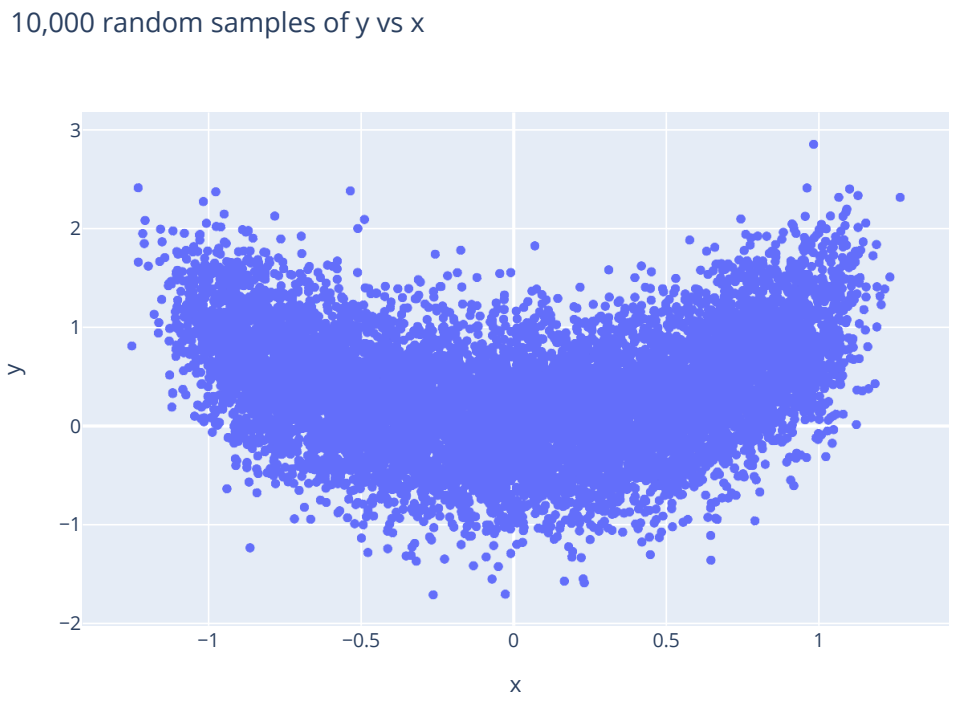
[8]:
```
plot_data = rng.choice(y, size=10000, replace=False)
px.histogram(plot_data, title="Histogram of 10,000 random values of y")
```



Loading [MathJax]/extensions/MathMenu.js

Now let's plot y against x:

[9]:
```
plot_idxes = rng.choice(n_tot, size=10000, replace=False)
px.scatter(x=x[plot_idxes], y=y[plot_idxes], title="10,000 random samples of y␣
↪vs x")
```



Nice banana shape!

σ will be constant and equal to 0.5 throughout this study:

[10]:
```
sigma = 0.5
```

6 Data subsample distribtution

Let's randomly assign $m = 10$ of the $n = 100$ first data points to the $a = 5$ agents.

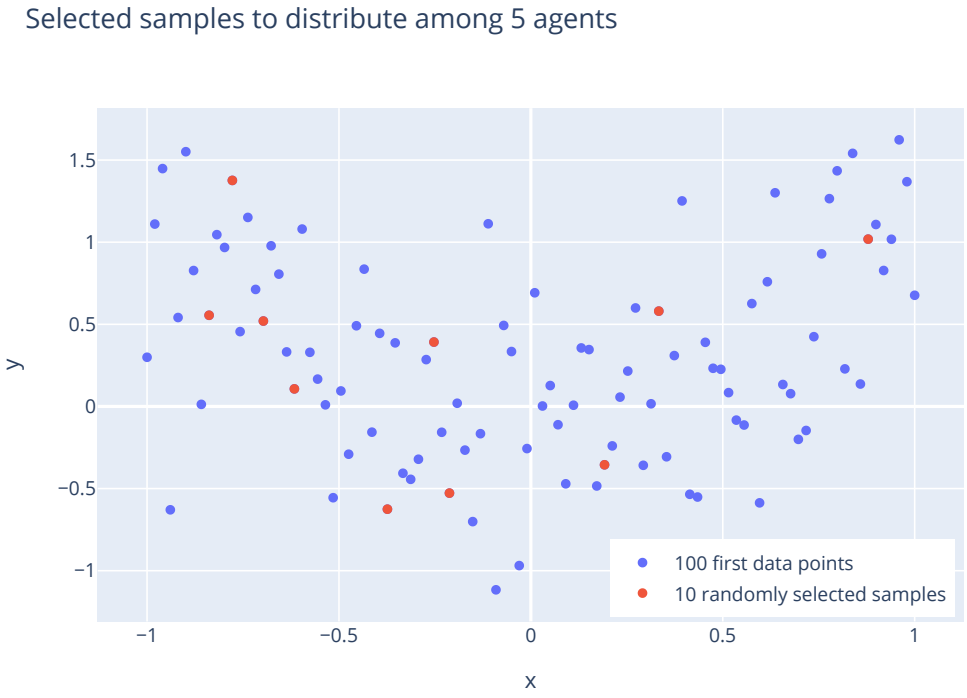
[11]:
```
n = 100  # pool size
m = ceil(sqrt(n))  # only m (scalar) data points *in total* given to the agents
a = 5  # number of agents

x_n, y_n, idx_sel_flat, idx_sel, x_sel_flat, x_sel, y_sel_flat, y_sel =␣
 ↪susbample_data(
    x, y, n, m, a, rng
)
```

```
n = 100
m = 10
a = 5
nb_points_per_agent = 2
```

```
Selected 10 points to distribute to the 5 agents among the 100 first data
points, resulting in 2 points per agent.
```

We can visualize the randomly selected samples (and check that we get back the orignal distribution when m is large enough):

[12]:
```
show_selected_points(x_n, y_n, n, x_sel_flat, y_sel_flat, m, a)
```



7 Communication graph

Let's now build the communication graph. You can choose any topology you want, though it might break convergence of course.
Some predefined W matrices are given below:

[13]:
```
### Choose W
# fully connected, undirected
W_0 = 1 / a * np.ones((a, a))
W = W_0.copy()

### Visualize W and check for symmetry and double stochasticity
print("Communication graph:")
```

```
print(W)
check_W(W)
```

```
Communication graph:
[[0.2 0.2 0.2 0.2 0.2]
 [0.2 0.2 0.2 0.2 0.2]
 [0.2 0.2 0.2 0.2 0.2]
 [0.2 0.2 0.2 0.2 0.2]
 [0.2 0.2 0.2 0.2 0.2]]
```

8 Kernel matrix

The kernel matrix K is defined as:

$$K = [k(x_i, x_j)]_{i,j}$$

where $k(x_i, x_j)$ is the kernel function defined as:

$$k(x_i, x_j) = \exp(-|x_i - x_j|^2)$$

K_{nm} and K_{mm} are furthermore defined as:

$$K_{nm} = [k(x_i, x_j)]_{i \leq n, j \in \mathcal{M}}$$
$$K_{mm} = [k(x_i, x_j)]_{i,j \in \mathcal{M}}$$

[14]:
```
K_nm, K_mm = build_kernel_matrices(x, x_sel_flat, n, m, idx_sel_flat)
```

9 Decentralized Gradient Descent

Let's implement and test the Decentralized Gradient Descent algorithm, where each local gradient for the agent k is:

$$\nabla f_k(\alpha_k) = \frac{1}{a} K_{mm} \alpha_k - \frac{1}{\sigma^2} K_{(k)m}^\top \left(y_{(k)} - K_{(k)m} \alpha_k \right)$$

where $y_{(k)}$ is the vector of the k-th agent's data points and $K_{(k)m}$ is the kernel matrix of the k-th agent's data points.

Note that K_{mm} is symmetric by construction.

9.1 Convergence conditions

9.1.1 Conditions

For a constant step size η, DGD converges if:

1. W is symmetric and doubly stochastic
2. $\gamma = \max |S_p(W)| \quad \{1\} < 1$, that is: the second-largest eigenvalue of W in absolute value is strictly smaller than 1
3. each f_k is convex and L_k-smooth

Then denoting $L = \max_k L_k$ and $\bar{\alpha} = \frac{1}{a}\sum_{k=1}^{a} \alpha_k$, we have, for $\eta \leq \mathcal{O}(1/L)$: - quasi-consensus:

$$\forall \text{ agent } k: \ ||\alpha_k - \bar{\alpha}|| \to \mathcal{O}\left(\frac{\eta}{1-\gamma}\right)$$

- quasi-convergence:

$$F^* - F(\bar{\alpha}) \to \mathcal{O}\left(\frac{\eta}{1-\gamma}\right)$$

where F is the global objective function.

9.1.2 Checks

1. was already checked above
2. let's check γ:

[15]:
```
eigenvals = np.linalg.eigvalsh(W)
assert np.allclose(1, eigenvals[-1])  # (re)check that W is row-stochastic
gamma = eigenvals[-2]
print("  ", round(gamma, 5))
assert gamma < 1
```

```
0.0
```

3. To check convexity and L-smoothness, let's compute the Hessian of f_k:

$$\nabla^2 f_k(\alpha_k) = \frac{1}{a} K_{mm} + \frac{1}{\sigma^2} K_{(k)m}^\top K_{(k)m} \tag{1}$$

$\nabla^2 f_k$ is clearly positive semi-definite, so f_k is convex. Furthermore, $\nabla^2 f_k$ is constant thus higher-bounded* by some $L_k I_m$, where I_m is the identity matrix of size m. Hence f_k is L_k-smooth.

*Here by bounded I mean $\nabla^2 f_k \preceq L_k I_m$, that is: $L_k I_m - \nabla^2 f_k$ is postive semi-definite.

Let's experimentally check this:

[16]:
```
Hessian_agents = np.zeros((a, m, m))
for k in range(a):
    K_agent_k = K_nm[idx_sel[k]]
    Hessian_agents[k] = 1 / a * K_mm + 1 / sigma**2 * K_agent_k.T @ K_agent_k

# check that all Hessian matrices are symmetric and positive definite
for k in range(a):
    assert (Hessian_agents[k].T == Hessian_agents[k]).all()
    assert np.all(np.linalg.eigvalsh(Hessian_agents[k]) > 0)

# find L_k as the maximum eigenvalue of the Hessian matrix
L = -np.inf
for k in range(a):
    eigenvals = np.linalg.eigvalsh(Hessian_agents[k])
    max_eig = eigenvals[-1]
    print("L_{} =".format(k), round(max_eig, 5))
    if max_eig > L:
```

```
        L = max_eig
print("\nL = ", round(L, 5))
```

```
L_0 = 55.4773
L_1 = 48.70668
L_2 = 42.6017
L_3 = 43.4164
L_4 = 27.31567

L =  55.4773
```

9.2 Hyperparameters

The important point here is that the step size has to be smaller than some $\mathcal{O}(1/L)$.

Experimentally, for this precise problem (and with $a = 5$), the constant hiding in this $\mathcal{O}$ seems to be around 1.

[17]:
```
print("1/L = {:.5E}".format(1 / L))
```

```
1/L = 1.80254E-02
```

[18]:
```
t_max = int(20e3)  # number of gradient iterations
step_size = 1e-2
```

9.3 Initialization

Let's distribute the alpha vectors to each agent; each alpha vector is randomly drawn from a uniform distribution between -1 and 1.

[19]:
```
alpha_agents_0 = 2 * rng.random((a, m)) - 1
```

9.4 DGD run

[20]:
```
alpha_seq = run_DGD(
    alpha_agents_0, t_max, a, K_mm, y_n, idx_sel, K_nm, sigma, m, W, step_size
)
```

```
  0%|          | 0/20000 [00:00<?, ?it/s]
```

9.5 Optimality gap

Let's plot the gap between the optimal value and the value of the agents. We will start by computing the optimal α^*.

The exact gradient of the total objective F:

$$F(\alpha) = \frac{1}{2}\alpha^\top K_{mm}\alpha + \frac{1}{2\sigma^2}\left\|y_n - K_{nm}\alpha\right\|_2^2$$

is:

$$\nabla F(\alpha) = K_{mm}\alpha - \frac{1}{\sigma^2}K_{nm}^\top\left(y_n - K_{nm}\alpha\right) \in \mathbb{R}^m$$

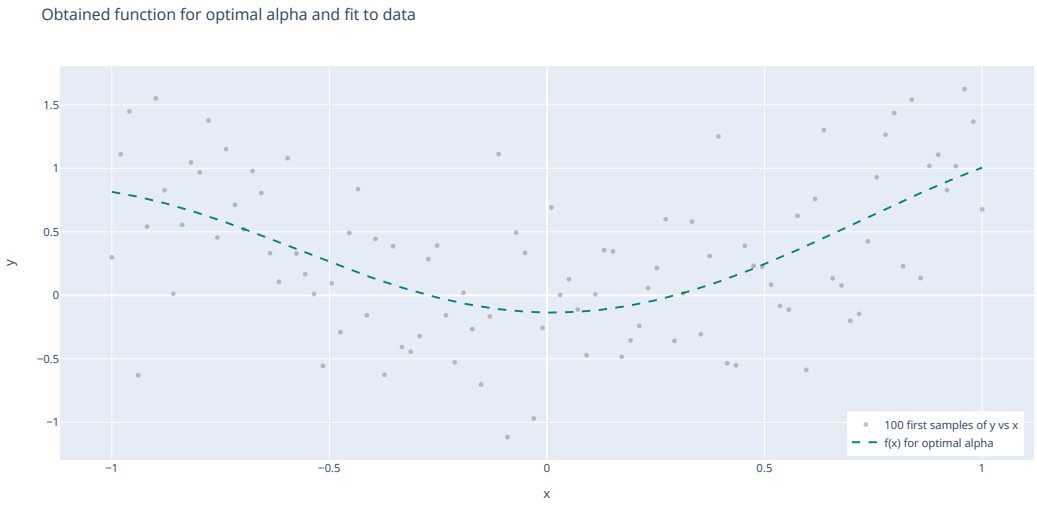
where y_n is the vector of the n first data points.

[21]:
```
y_n = y[:n]
# The below expression comes from expressing the nullity of the gradient of the␣
 ↪objective at extrema
alpha_opti = np.linalg.solve(
    K_mm + 1 / sigma**2 * K_nm.T @ K_nm, 1 / sigma**2 * K_nm.T @ y_n
)
print("Real optimal alpha:", alpha_opti)
```

```
Real optimal alpha: [-3.72745607e+03  4.20190640e+03 -5.67413138e+03
-8.13511508e+02
 -5.42985418e+03  1.90155935e+02  8.25972296e+03 -9.43516909e+01
  3.08305064e+03  6.02120686e+00]
```

Let's see how it fits to the data:

[22]:
```
x_prime = np.linspace(-1, 1, 1000)

plot_f(n, x_n, y, x_prime, alpha_opti, None, x_sel_flat)
```

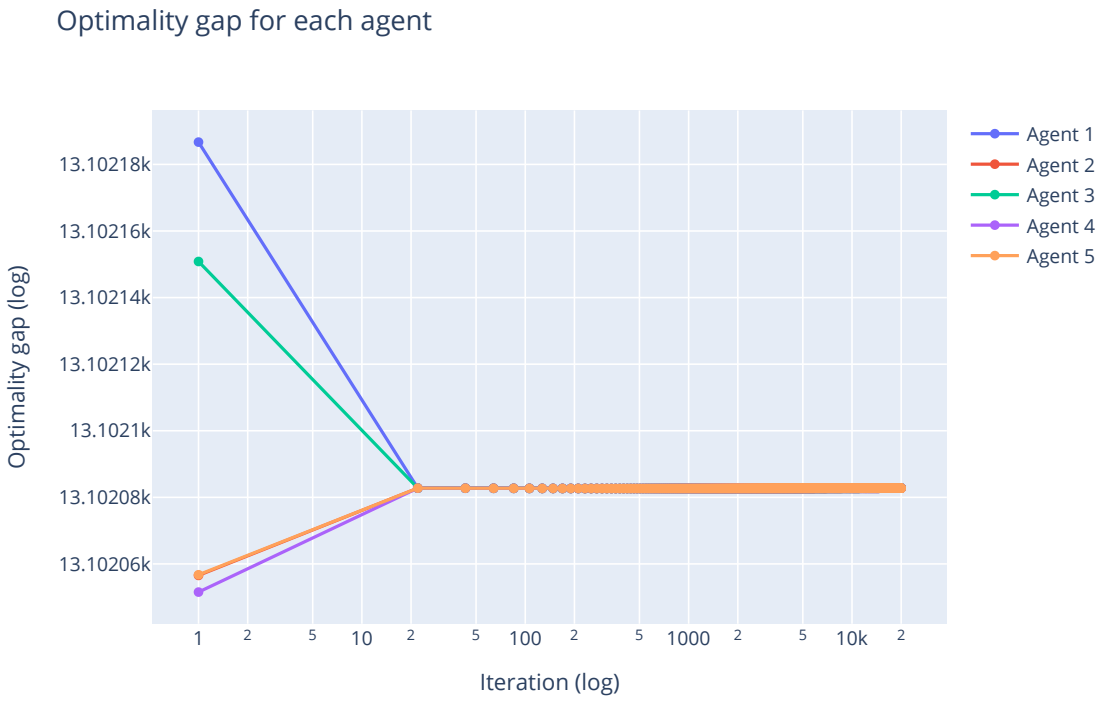


Not bad!

Now let's have a look at the optimality gaps:

[23]:
```
plot_opt_gap_per_agent(a, t_max, alpha_opti, alpha_seq)
```

```
Warning: t_max > 1000, plotting only 1000 points.
```



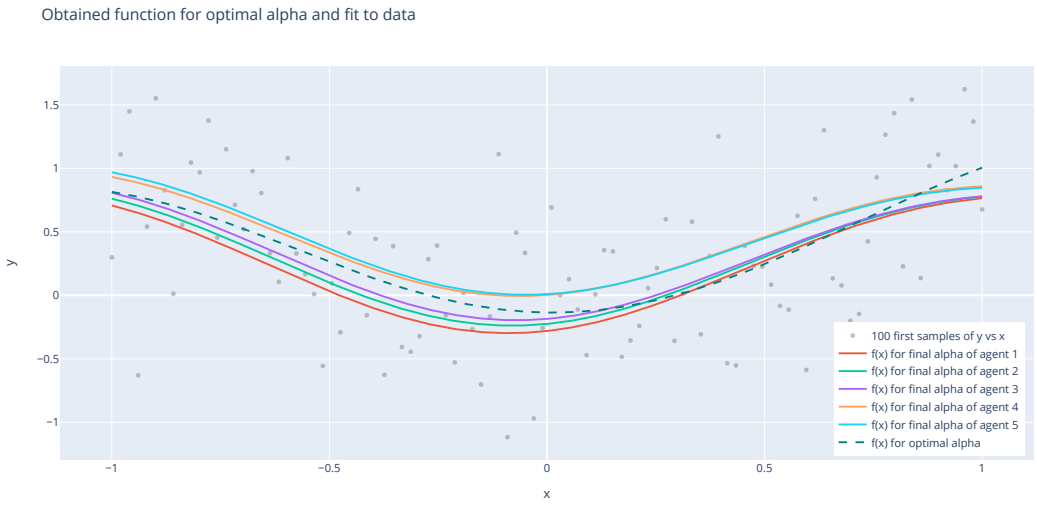
Note that zooming on the curve might be necessary if the first values are too far appart.

The DGD algorithm seems to stall after a "length" of around 100 (that is: $10,000$ iterations with a `step_size` of 10^{-2}, or $100,000$ iterations with a `step_size` of 10^{-3}).

At equal iteration length (number of iterations $\times$ step size), the consensus seems better when the step size is smaller.

9.6 Obtained function

[24]:
```
x_prime = np.linspace(-1, 1, 1000)

plot_f(n, x_n, y_n, x_prime, alpha_opti, alpha_seq[-1], x_sel_flat)
```



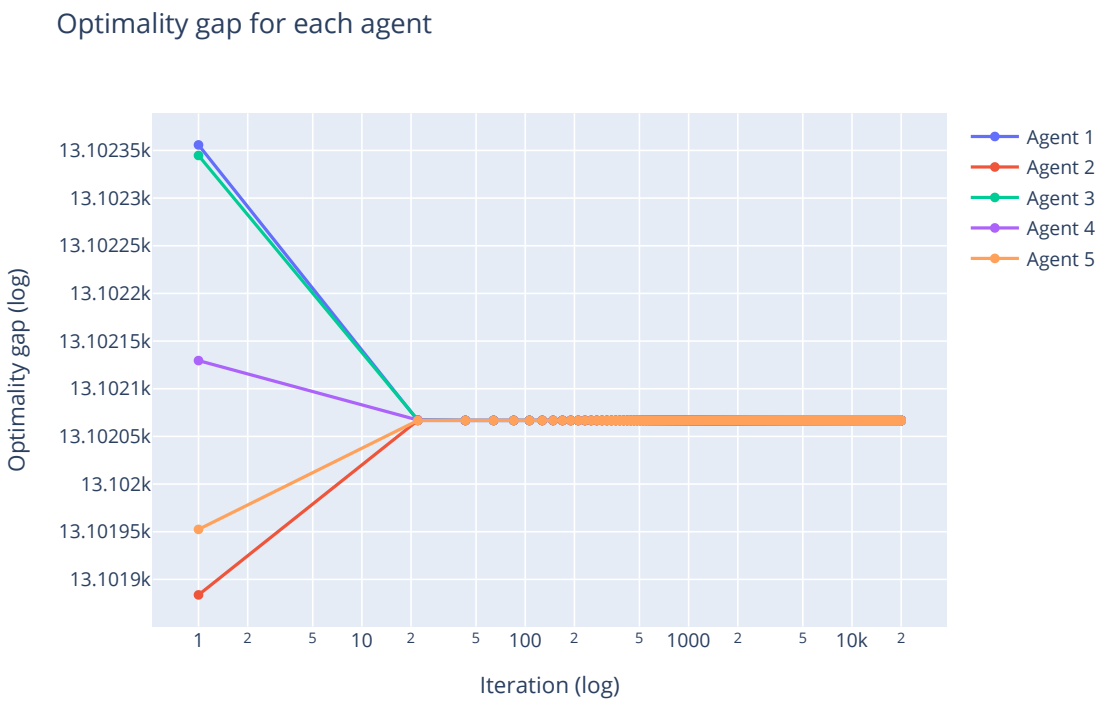
9.7 Variations

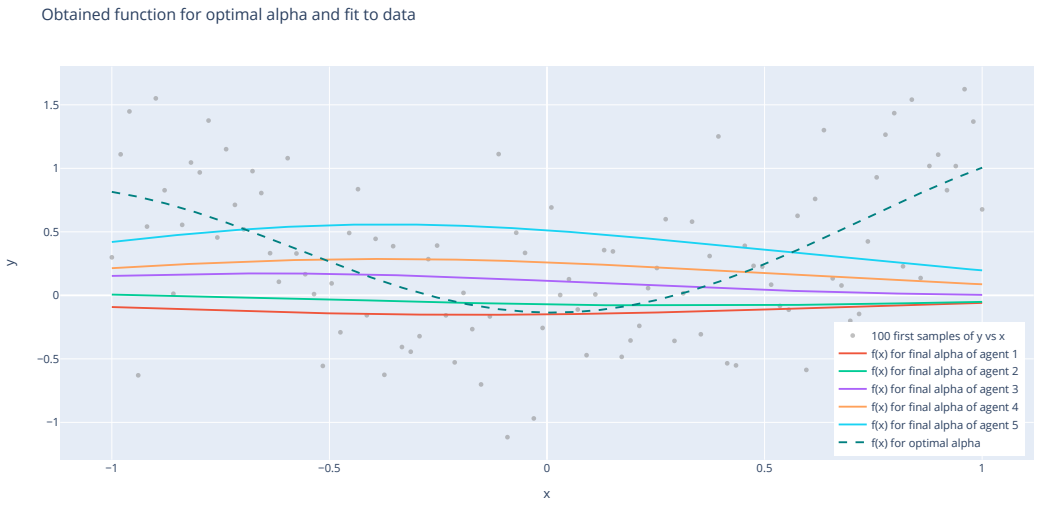
Let's change W to see how it affects the convergence.

[25]:
```
### Choose W
W_1 = np.array(
    [
        [0.5, 0.2, 0.0, 0.0, 0.0],
        [0.2, 0.5, 0.0, 0.0, 0.0],
        [0.0, 0.0, 0.5, 0.3, 0.0],
        [0.0, 0.0, 0.3, 0.5, 0.0],
        [0.0, 0.0, 0.0, 0.0, 0.5],
    ]
)
W = W_1.copy()
### Visualize W and check for symmetry and double stochasticity
print("Communication graph:")
print(W)
# check_W(W) # W is not doubly stochastic!

eigenvals = np.linalg.eigvalsh(W)
# assert np.allclose(1, eigenvals[-1])  # W is not row-stochastic!
gamma = eigenvals[-2]
print("  ", round(gamma, 5))
assert gamma < 1

### Run DGD
alpha_seq = run_DGD(
    alpha_agents_0, t_max, a, K_mm, y_n, idx_sel, K_nm, sigma, m, W, step_size
```

```
)
### Plots
plot_opt_gap_per_agent(a, t_max, alpha_opti, alpha_seq)
plot_f(n, x_n, y_n, x_prime, alpha_opti, alpha_seq[-1], x_sel_flat)
```

```
Communication graph:
[[0.5 0.2 0.  0.  0. ]
 [0.2 0.5 0.  0.  0. ]
 [0.  0.  0.5 0.3 0. ]
 [0.  0.  0.3 0.5 0. ]
 [0.  0.  0.  0.  0.5]]
   0.7

  0%|          | 0/20000 [00:00<?, ?it/s]

Warning: t_max > 1000, plotting only 1000 points.
```



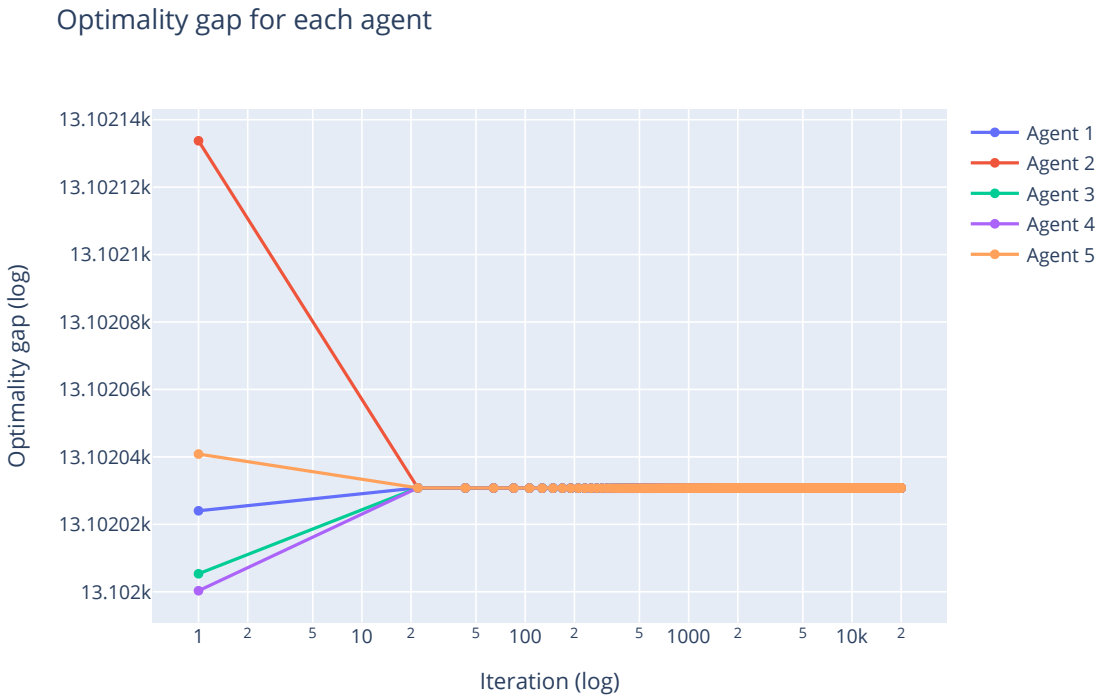


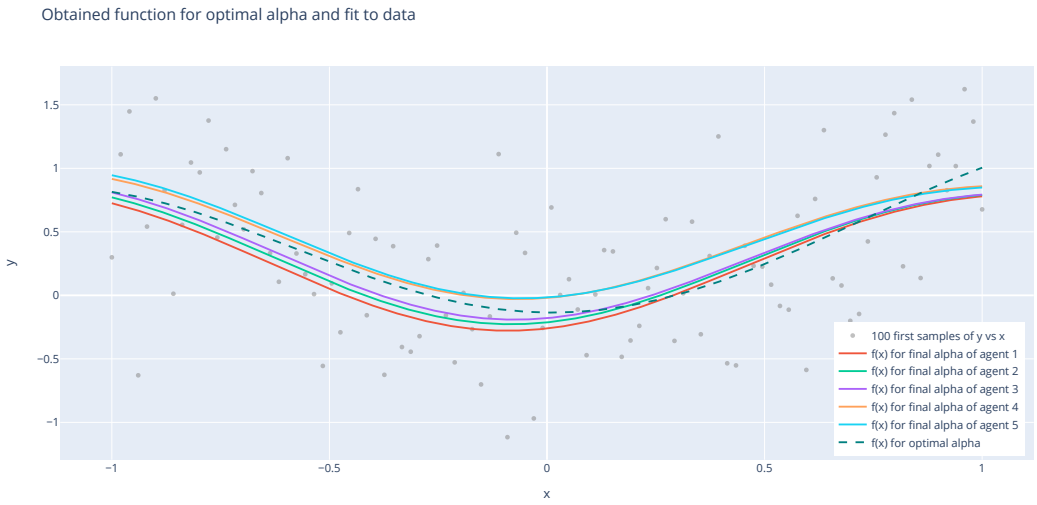
As expected, DGD does not (quasi-)converge anymore.
Let's try another W:

[26]:
```
### Choose W
W_2 = np.array(
    [
        [0., 0.25, 0.25, 0.25, 0.25],
        [0.25, 0., 0.25, 0.25, 0.25],
        [0.25, 0.25, 0., 0.25, 0.25],
        [0.25, 0.25, 0.25, 0., 0.25],
        [0.25, 0.25, 0.25, 0.25, 0.],
    ]
)
W = W_2.copy()
# Visualize W and check for symmetry and double stochasticity
print("Communication graph:")
print(W)
check_W(W) # W is doubly stochastic!

eigenvals = np.linalg.eigvalsh(W)
assert np.allclose(1, eigenvals[-1])  # W is row-stochastic!
gamma = eigenvals[-2]
print("  ", round(gamma, 5))
assert gamma < 1

### Run DGD
alpha_seq = run_DGD(
    alpha_agents_0, t_max, a, K_mm, y_n, idx_sel, K_nm, sigma, m, W, step_size
)
```

```
### Plots
plot_opt_gap_per_agent(a, t_max, alpha_opti, alpha_seq)
plot_f(n, x_n, y_n, x_prime, alpha_opti, alpha_seq[-1], x_sel_flat)
```

```
Communication graph:
[[0.   0.25 0.25 0.25 0.25]
 [0.25 0.   0.25 0.25 0.25]
 [0.25 0.25 0.   0.25 0.25]
 [0.25 0.25 0.25 0.   0.25]
 [0.25 0.25 0.25 0.25 0.  ]]
  -0.25

  0%|          | 0/20000 [00:00<?, ?it/s]

Warning: t_max > 1000, plotting only 1000 points.
```





Double stochasticity seems to really be necessary to guarantee (quasi-)convergence!

10 Gradient tracking

Let's implement and test the Gradient tracking algorithm.

10.1 Convergence conditions

10.1.1 Conditions

The convergence conditions are the same as for DGD, with two additions:

4. W has to be of positives coefficients
5. the f_k's need to be strongly convex

Then, with the same notations, we have:

- consensus:
$$\forall \text{ agent } k: \ ||\alpha_k - \bar{\alpha}|| \to 0$$
- convergence:
$$||\alpha^* - \bar{\alpha}|| \to 0$$

10.1.2 Checks

All conditions shared with the DGD results still hold.

1. is true with the default proposed W (W_0), but note that it must be checked, as symmetric and doubly stochastic matrices are not necessarily of positive coefficients.

[27]:
```
W = W_0.copy()
```

[28]:
```
# Visualize W and check for symmetry and double stochasticity
print("Communication graph:")
print(W)
check_W(W) # W is doubly stochastic!

eigenvals = np.linalg.eigvalsh(W)
assert np.allclose(1, eigenvals[-1])  # W is row-stochastic!
gamma = eigenvals[-2]
print("  ", round(gamma, 5))
assert gamma < 1
```

```
Communication graph:
[[0.2 0.2 0.2 0.2 0.2]
 [0.2 0.2 0.2 0.2 0.2]
 [0.2 0.2 0.2 0.2 0.2]
 [0.2 0.2 0.2 0.2 0.2]
 [0.2 0.2 0.2 0.2 0.2]]
   0.0
```

[29]:
```
assert (W >= 0).all()
```

1. results from Equation (1) showing that $\nabla^2 f_k$ is constant, thus lower-bounded by some $\omega_k I_m$.

10.2 Hyperparameters

Gradient tracking seems to necessitate a smaller step size than DGD.

[30]:
```
t_max = int(10e3)  # number of gradient iterations
step_size = 1e-3
```

10.3 Initialization

Same as for the DGD algorithm, but this time an initial gradient estimate is also needed for each agent.

I will initialize the gradient estimates to 0.

[31]:
```
alpha_agents_0 = 2 * rng.random((a, m)) - 1
grad_agents_0 = np.zeros((a, m))
```

10.4 GT run

[32]:
```
alpha_seq, grad_seq = run_GT(
    alpha_agents_0,
    grad_agents_0,
    t_max,
    a,
    K_mm,
    y_n,
    idx_sel,
```

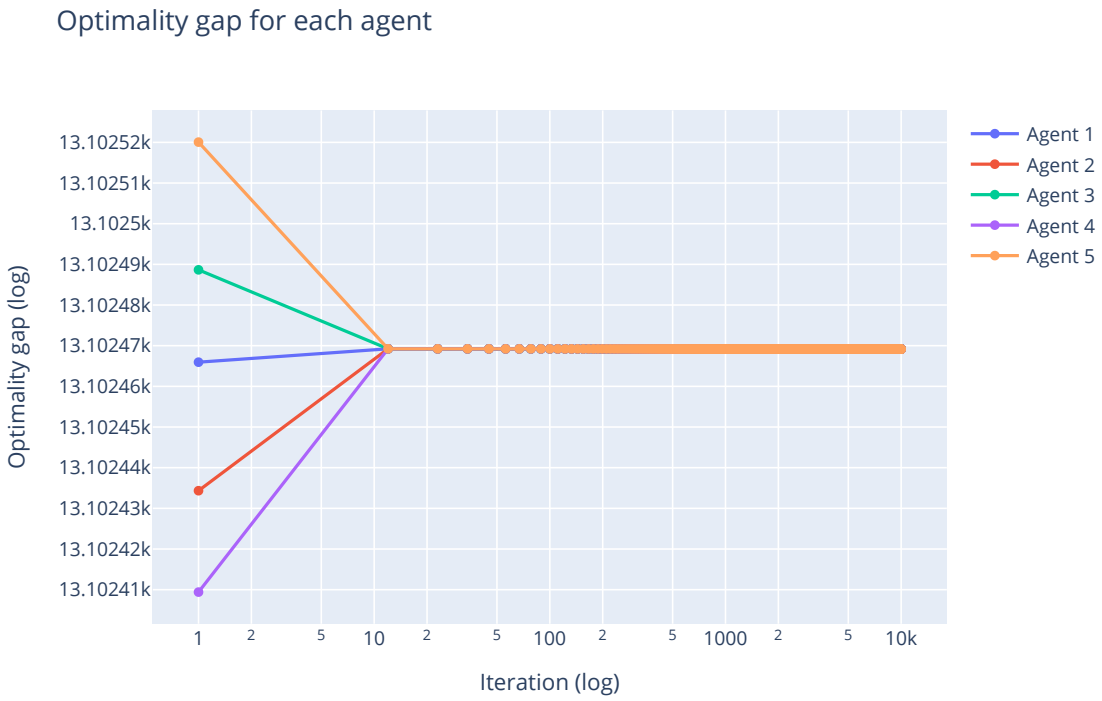
```
    K_nm,
    sigma,
    m,
    W,
    step_size,
)
```

```
  0%|          | 0/10000 [00:00<?, ?it/s]
```

10.5 Optimality gap

[33]:
```
plot_opt_gap_per_agent(a, t_max, alpha_opti, alpha_seq)
```

```
Warning: t_max > 1000, plotting only 1000 points.
```



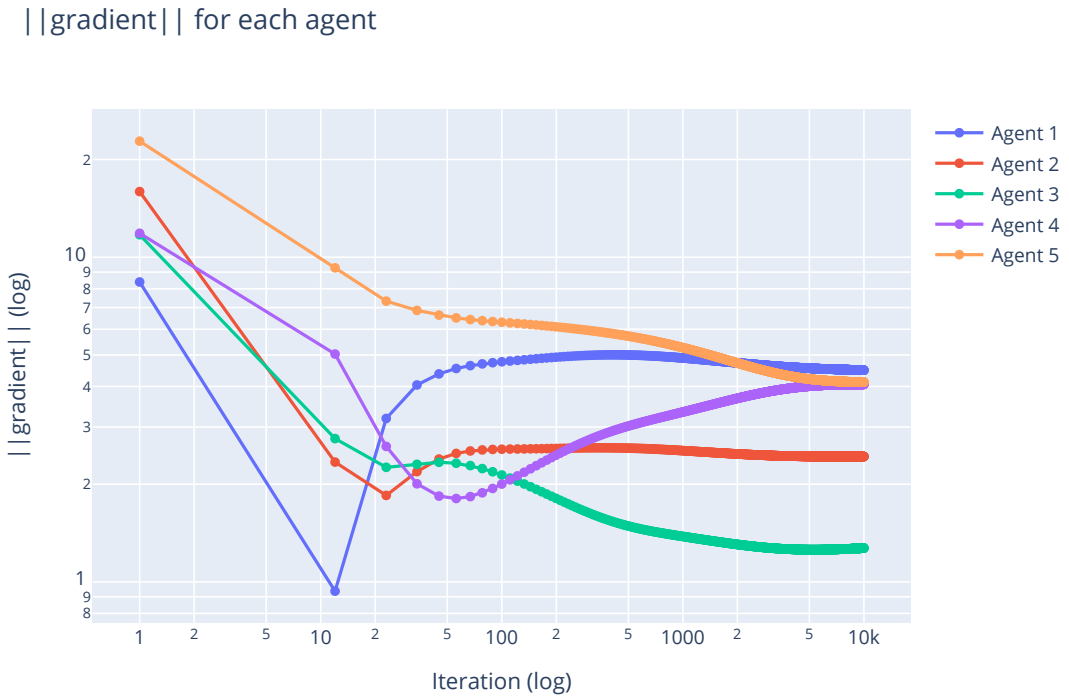
I am quite surprised as GT does not seem to yield better performances than DGD, while its convergence is supposed to be linear.

In particular, *it does not seem to be converging (exactly to the optimum)!*

10.6 Gradients

[34]:
```
print_vector_norms(a, t_max, grad_seq, "gradient")
```

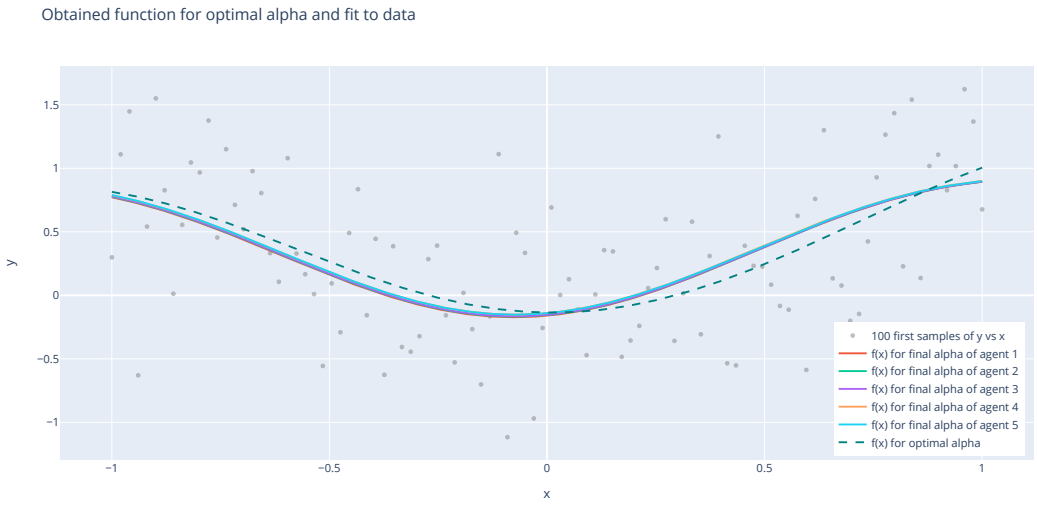
```
Skipping first iteration for x log plot.
Warning: t_max > 1000, plotting only 1000 points.
```



Weirdly, the gradients do not seem to converge towards zero (at least not too hastily)...

10.7 Obtained function

[35]:
```
plot_f(n, x_n, y_n, x_prime, alpha_opti, alpha_seq[-1], x_sel_flat)
```



The obtained function is closer to the "real" one than for DGD however.

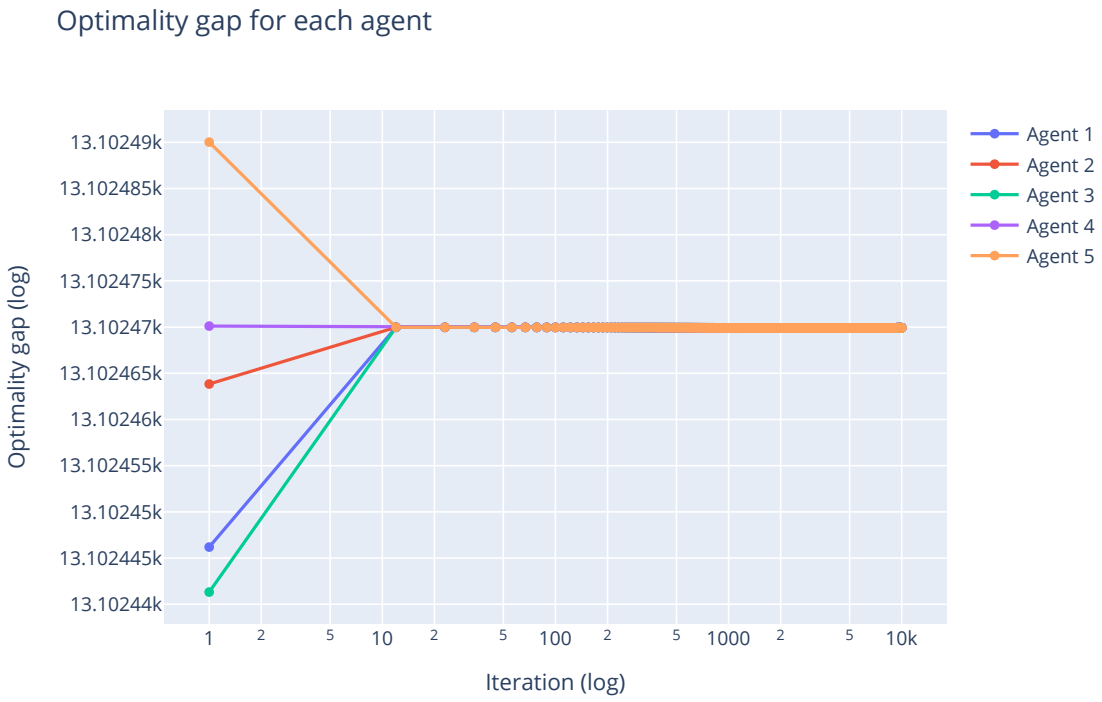
10.8 Variations

If we vary a bit W, GT still converges:

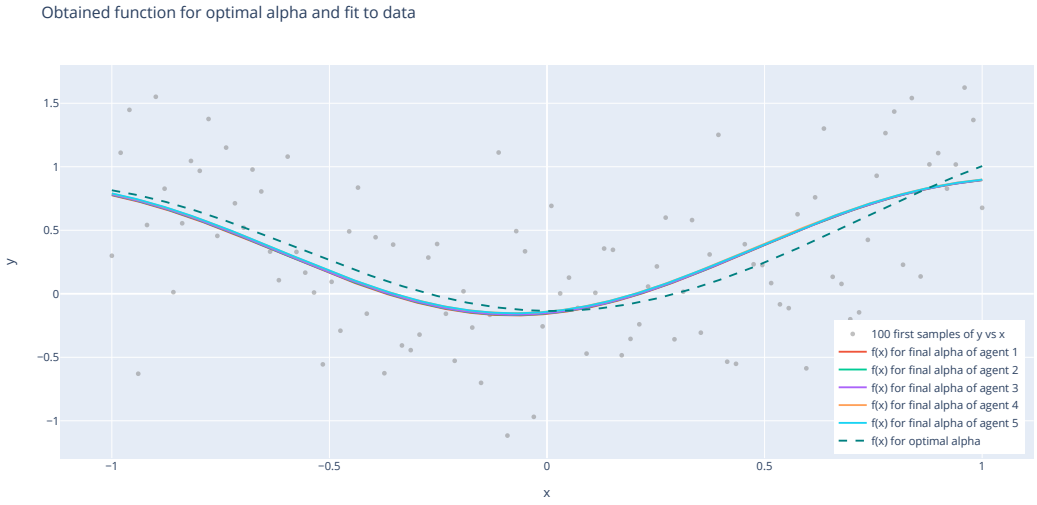
[36]:
```
### Choose W
W_3 = np.array(
    [
        [0.1, 0.225, 0.225, 0.225, 0.225],
        [0.225, 0.1, 0.225, 0.225, 0.225],
        [0.225, 0.225, 0.1, 0.225, 0.225],
        [0.225, 0.225, 0.225, 0.1, 0.225],
        [0.225, 0.225, 0.225, 0.225, 0.1],
    ]
)
W = W_3.copy()
# Visualize W and check for symmetry and double stochasticity
print("Communication graph:")
print(W)
check_W(W) # W is doubly stochastic!
eigenvals = np.linalg.eigvalsh(W)
assert np.allclose(1, eigenvals[-1])  # W is row-stochastic!
gamma = eigenvals[-2]
print("  ", round(gamma, 5))
assert gamma < 1
assert (W >= 0).all()
### Run GT
alpha_seq, grad_seq = run_GT(
```

```
    alpha_agents_0,
    grad_agents_0,
    t_max,
    a,
    K_mm,
    y_n,
    idx_sel,
    K_nm,
    sigma,
    m,
    W,
    step_size,
)
### Plots
plot_opt_gap_per_agent(a, t_max, alpha_opti, alpha_seq)
print_vector_norms(a, t_max, grad_seq, "gradient")
plot_f(n, x_n, y_n, x_prime, alpha_opti, alpha_seq[-1], x_sel_flat)
```

```
Communication graph:
[[0.1   0.225 0.225 0.225 0.225]
 [0.225 0.1   0.225 0.225 0.225]
 [0.225 0.225 0.1   0.225 0.225]
 [0.225 0.225 0.225 0.1   0.225]
 [0.225 0.225 0.225 0.225 0.1  ]]
   -0.125

  0%|          | 0/10000 [00:00<?, ?it/s]

Warning: t_max > 1000, plotting only 1000 points.
```



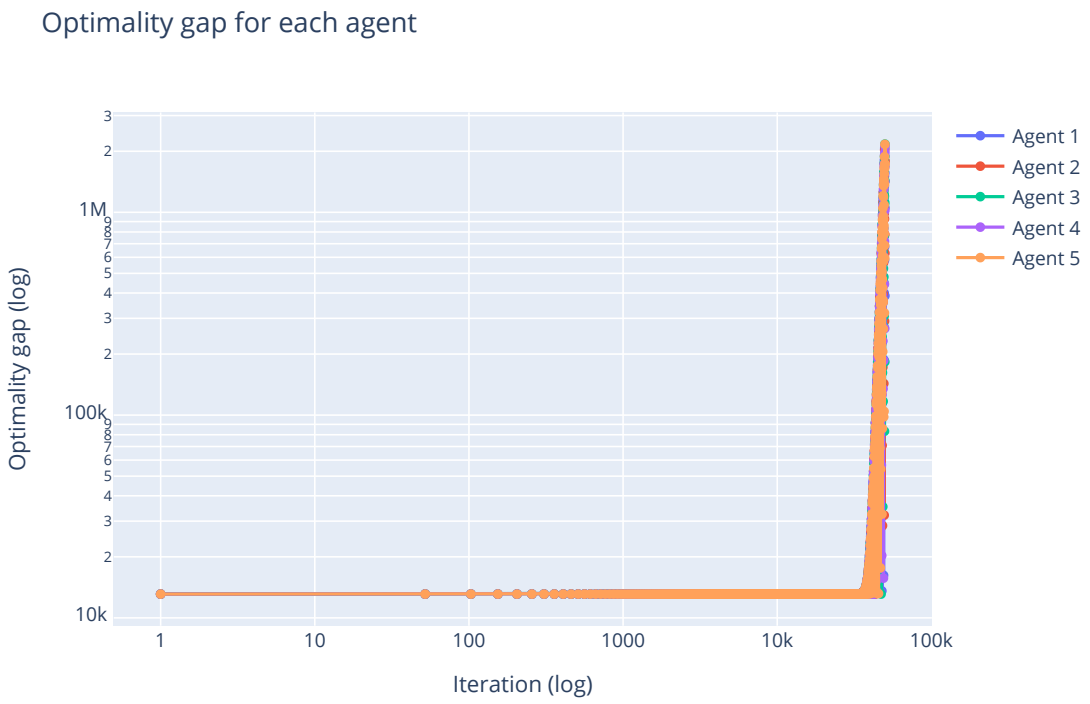

```
Skipping first iteration for x log plot.
Warning: t_max > 1000, plotting only 1000 points.
```



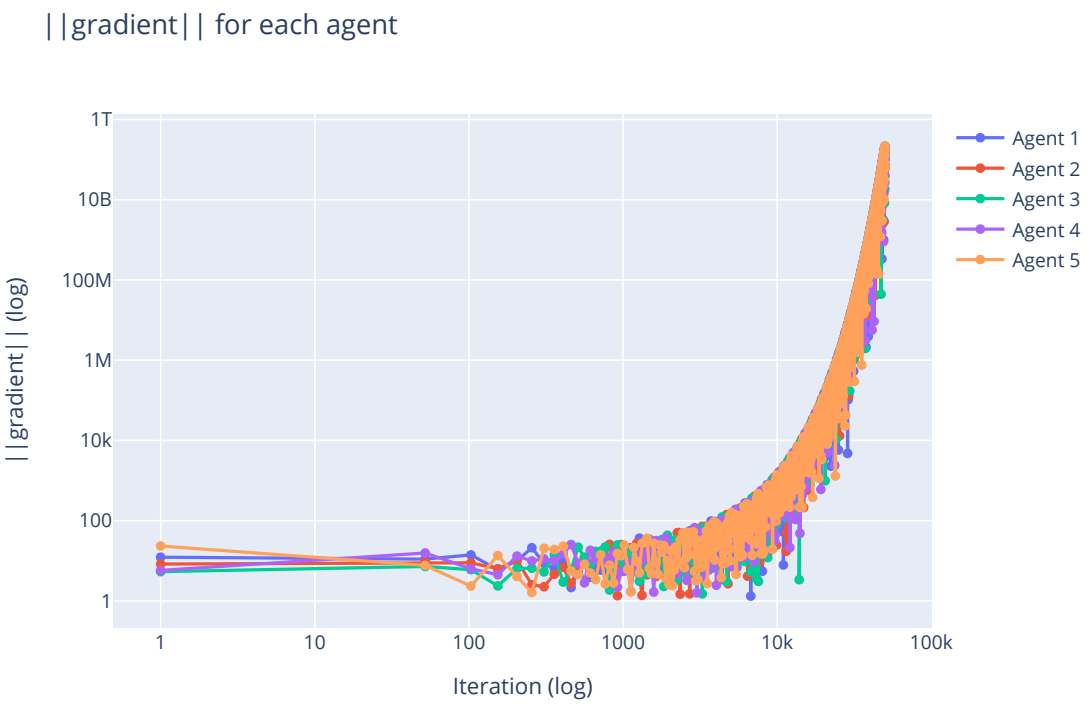
But even with a very small step size, if $W_{kk} = 0$ for some k, GT does not converge anymore:

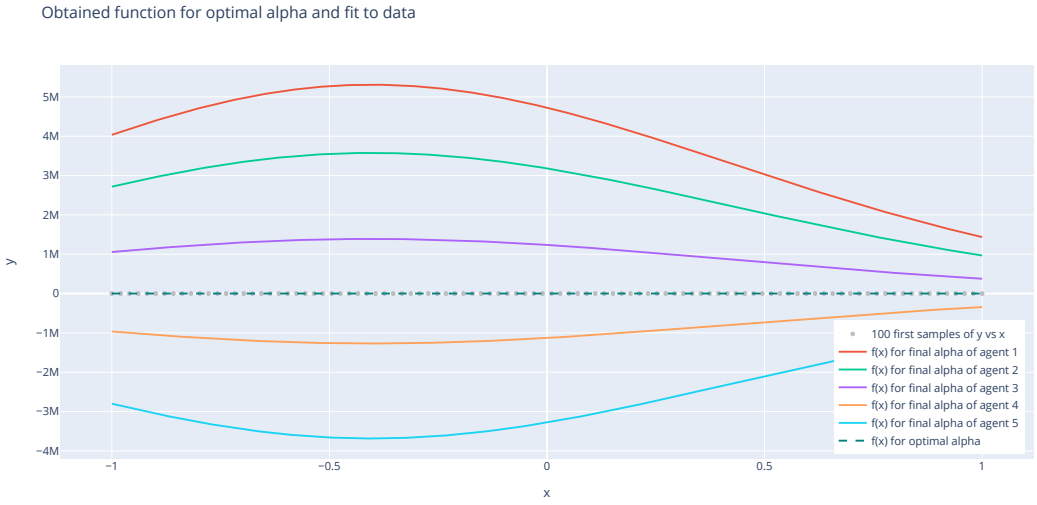
[37]:
```
### Choose W
W = W_2.copy()
# Visualize W and check for symmetry and double stochasticity
print("Communication graph:")
print(W)
check_W(W) # W is doubly stochastic!
eigenvals = np.linalg.eigvalsh(W)
assert np.allclose(1, eigenvals[-1])  # W is row-stochastic!
gamma = eigenvals[-2]
print("  ", round(gamma, 5))
assert gamma < 1
assert (W >= 0).all()
### Hyperparameters
t_max = int(50e3)
step_size = 1e-5
### Run GT
alpha_seq, grad_seq = run_GT(
    alpha_agents_0,
    grad_agents_0,
    t_max,
    a,
    K_mm,
    y_n,
    idx_sel,
    K_nm,
    sigma,
    m,
    W,
    step_size,
)
### Plots
plot_opt_gap_per_agent(a, t_max, alpha_opti, alpha_seq)
print_vector_norms(a, t_max, grad_seq, "gradient")
plot_f(n, x_n, y_n, x_prime, alpha_opti, alpha_seq[-1], x_sel_flat)
```

```
Communication graph:
[[0.   0.25 0.25 0.25 0.25]
 [0.25 0.   0.25 0.25 0.25]
 [0.25 0.25 0.   0.25 0.25]
 [0.25 0.25 0.25 0.   0.25]
 [0.25 0.25 0.25 0.25 0.  ]]
   -0.25

  0%|          | 0/50000 [00:00<?, ?it/s]

Warning: t_max > 1000, plotting only 1000 points.
```




```
Skipping first iteration for x log plot.
Warning: t_max > 1000, plotting only 1000 points.
```


||gradient|| for each agent
1T
10B
100M
1M
10k
100
1
||gradient|| (log)
1
10
100
1000
10k
100k
Iteration (log)
Agent 1
Agent 2
Agent 3
Agent 4
Agent 5


Obtained function for optimal alpha and fit to data
5M
4M
3M
2M
1M
0
−1M
−2M
−3M
−4M
y
−1
−0.5
0
0.5
1
x
100 first samples of y vs x
f(x) for final alpha of agent 1
f(x) for final alpha of agent 2
f(x) for final alpha of agent 3
f(x) for final alpha of agent 4
f(x) for final alpha of agent 5
f(x) for optimal alpha

[38]:
```
W = W_0.copy()
```

11 Dual Decomposition

11.1 Reformulation

In order to implement the dual decomposition algorithm, we need to define the dual problem.

The initial problem is:

$$\arg\min_{\alpha\in\mathbb{R}^m} F(\alpha) = \arg\min_{\alpha\in\mathbb{R}^m} \frac{1}{2}\alpha^\top K_{mm}\alpha + \frac{1}{2\sigma^2}\left\|y_n - K_{nm}\alpha\right\|_2^2$$

which we rewrite in the context of distributed optimization as:

$$\arg\min_{\alpha\in\mathbb{R}^m} \sum_{k=1}^{a} f_k(\alpha) = \arg\min_{\alpha\in\mathbb{R}^m} \sum_{k=1}^{a} \left[\frac{1}{a}\frac{1}{2}\alpha^\top K_{mm}\alpha + \frac{1}{2\sigma^2}\left\|y_{(k)} - K_{(k)m}\alpha\right\|_2^2\right]$$

We can then reformulate this problem as:

$$\arg\min_{\substack{\alpha_k\in\mathbb{R}^m \\ 1\le k\le a}} \sum_{k=1}^{a} f_k(\alpha_k) \quad \text{subject to} \quad \alpha_k = \alpha_p \quad \text{if} \quad W_{kp}\neq 0$$

which is equivalent to:

$$\arg\min_{\alpha\in\mathbb{R}^{am}} F(\alpha) \quad \text{subject to} \quad A\alpha = 0$$

where: - $F(\alpha) = \sum_{k=1}^{a} f_k(\alpha_k = \alpha_{[(k-1)m\,:\,km]})$ - $A \in \mathcal{M}_{d\times a}(\mathbb{R})$ is defined by concatenating rows of zeros of dimension a with a 1 in position k and a -1 in position p for each pair (k,p) such that $W_{kp}\neq 0$ and $k<p$ (I reversed the order of the indices w.r.t. the slides). In the following, A_{kp} will ***not*** denote the (k,p)-th element of A, but:

$$A_{kp} = \begin{cases} 1 & \text{if } k\sim p \text{ and } k<p \\ -1 & \text{if } k\sim p \text{ and } k>p \\ 0 & \text{otherwise} \end{cases} \quad \in\mathbb{R}$$

d is thus "dynamically" defined for each problem, depending on the graph topology. Note that we could try to *optimize* this dimension, which typically grows as m^2 ($m(m-1)/2$ precisely) with the naive approach used here, but this is a problem *per se* :-)

11.2 Dual algorithm

This can finally be rewritten by dualizing and distributing a copy α_k of α to each agent k as the following *separated* problems:

for each agent k, solve:
$$\begin{aligned}
\alpha_k^*(\lambda) &= \arg\min_{\alpha_k\in\mathbb{R}^m} L_k(\alpha_k,\lambda) \\
&= \arg\min_{\alpha_k\in\mathbb{R}^m} \left[f_k(\alpha_k) + \sum_{p\sim k}\lambda_{kp}^\top A_{kp}\alpha_k\right] \\
&= \arg\min_{\alpha_k\in\mathbb{R}^m} \left[\frac{1}{a}\frac{1}{2}\alpha_k^\top K_{mm}\alpha_k + \frac{1}{2\sigma^2}\left\|y_{(k)} - K_{(k)m}\alpha_k\right\|_2^2 + \sum_{p\sim k}\lambda_{kp}^\top A_{kp}\alpha_k\right]
\end{aligned}$$

where $\lambda_{kp} \in \mathbb{R}^m$ is the dual variable associated with the constraint $\alpha_k = \alpha_p$.

Then follows a distributed dual update of the dual variables:

$$\lambda_{kp} \leftarrow \lambda_{kp} + \eta \left(\alpha_k^*(\lambda) - \alpha_p^*(\lambda)\right)$$

Each agent computes locally its own α_k^*. I will solve this problem by an exact method, that is: by finding the stationary point of the Lagrangian $L_k(\alpha_k, \lambda)$ with respect to α_k:

$$\begin{aligned} 0 &= \nabla_{\alpha_k} L_k(\alpha_k, \lambda) \\ &= \nabla f_k(\alpha_k) + \sum_{p \sim k} A_{kp}\lambda_{kp} \\ &= \frac{1}{a} K_{mm}\alpha_k - \frac{1}{\sigma^2} K_{(k)m}^\top \left(y_{(k)} - K_{(k)m}\alpha_k\right) + \sum_{p \sim k} A_{kp}\lambda_{kp} \end{aligned}$$

11.3 Convergence conditions

The Dual Decomposition algorithm converges if, as for Gradient Tracking, the f_k's are ω_k-strongly convex and Lipschitz-smooth, which still holds.

Denoting $\omega = \max_k \omega_k$, one can then select $\eta < 2\omega/\sigma_{\max}(A)^2$, with $\sigma_{\max}(A)$ the largest singular value of A, and obtain linear convergence of both the dual variables and the primal variables to the optimal solution.

11.4 Checks

Let's hence compute ω and $\sigma_{\max}(A)$.

f_k is ω_k-strongly convex if and only if $\nabla^2 f_k(\alpha_k) \succeq \omega_k I_m$, that is: iff $\nabla^2 f_k(\alpha_k) - \omega_k I_m$ is positive semi-definite.

[39]:
```
# find _k as the minimum eigenvalue of the Hessian matrix
omega = np.inf
for k in range(a):
    # the Hessian is symmetric
    eigenvals = np.linalg.eigvalsh(Hessian_agents[k])
    min_eig = eigenvals[0]
    print(f" _{k} = {min_eig}")
    if min_eig < omega:
        omega = min_eig
print("\n  = ", omega)
```

```
 _0 = 4.11746857055975e-13
 _1 = 4.099100549589876e-13
 _2 = 4.108870080138924e-13
 _3 = 4.0619634300758933e-13
 _4 = 4.0901700546370697e-13

  =  4.0619634300758933e-13
```

Let's now compte $\sigma_{\max}(A)$

[40]:
```
A = build_A(W, a)
print("A:\n", A)
print("\nshape:", A.shape)
```

```
A:
 [[ 1. -1.  0.  0.  0.]
 [ 1.  0. -1.  0.  0.]
 [ 1.  0.  0. -1.  0.]
 [ 1.  0.  0.  0. -1.]
 [ 0.  1. -1.  0.  0.]
 [ 0.  1.  0. -1.  0.]
 [ 0.  1.  0.  0. -1.]
 [ 0.  0.  1. -1.  0.]
 [ 0.  0.  1.  0. -1.]
 [ 0.  0.  0.  1. -1.]]

shape: (10, 5)
```

[41]:
```
svdvals = np.linalg.svd(A, full_matrices=False, compute_uv=False)
sigma_min = np.min(svdvals)
sigma_max = np.max(svdvals)
print(" _max:", round(sigma_max, 5))
```

```
 _max: 2.23607
```

[42]:
```
bound = 2 * omega / sigma_max**2
print(f"Theoretical upper bound on step size  : {bound:.1E}")
```

```
Theoretical upper bound on step size  : 1.6E-13
```

Well that is pretty small...

11.5 Hyperparameters

[43]:
```
t_max = int(1e5)  # number of gradient iterations
step_size = 1e-15
```

11.6 Initialization

This time an additional initial dual variable λ is needed for each agent.

Because A may very well not be full row-rank, we might need to select λ_0 in the image of A. 0 is a safe choice!

[44]:
```
print(" _min:", sigma_min)
if sigma_min == 0:
    print("A is not full row-rank!  _0 must be chosen in the image of A.")
```

```
 _min: 2.514140425170977e-16
```

[45]:
```
alpha_agents_0 = 2 * rng.random((a, m)) - 1
# alpha_agents_0 = np.zeros((a, m))

lambda_agents_0 = np.zeros((a, a, m))
```

11.7 DD run

[46]:
```
alpha_seq, lambda_seq = run_DD(
    alpha_agents_0,
    lambda_agents_0,
    t_max,
    a,
    K_mm,
    y_n,
    idx_sel,
    K_nm,
    sigma,
    m,
    W,
    step_size,
    beta_penalize=None,  # penalize with /2 || ||_2^2
)
```

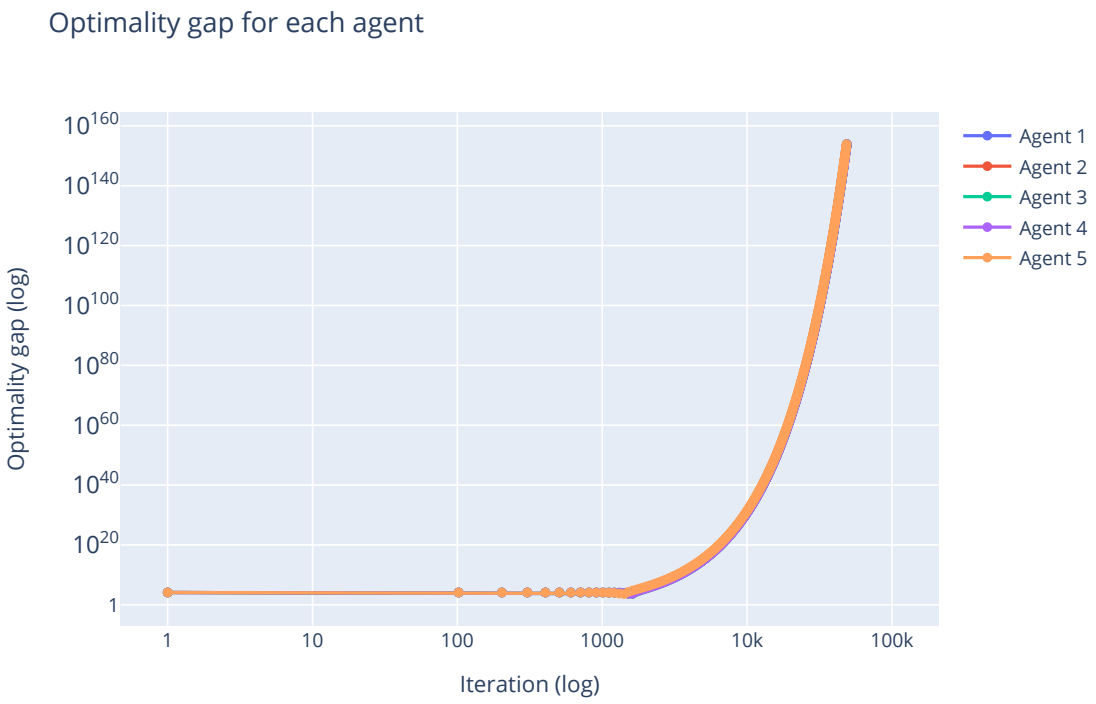
```
  0%|          | 0/100000 [00:00<?, ?it/s]
```

```
/home/warpig/OneDrive/Travail/ENSTA/3A/Cours
3A/SOD314_Cooperative_Opti/project/src/algs.py:148: RuntimeWarning:

invalid value encountered in subtract
```

11.8 Optimality gap

[47]:
```
plot_opt_gap_per_agent(a, t_max, alpha_opti, alpha_seq)
```

```
Warning: t_max > 1000, plotting only 1000 points.
```

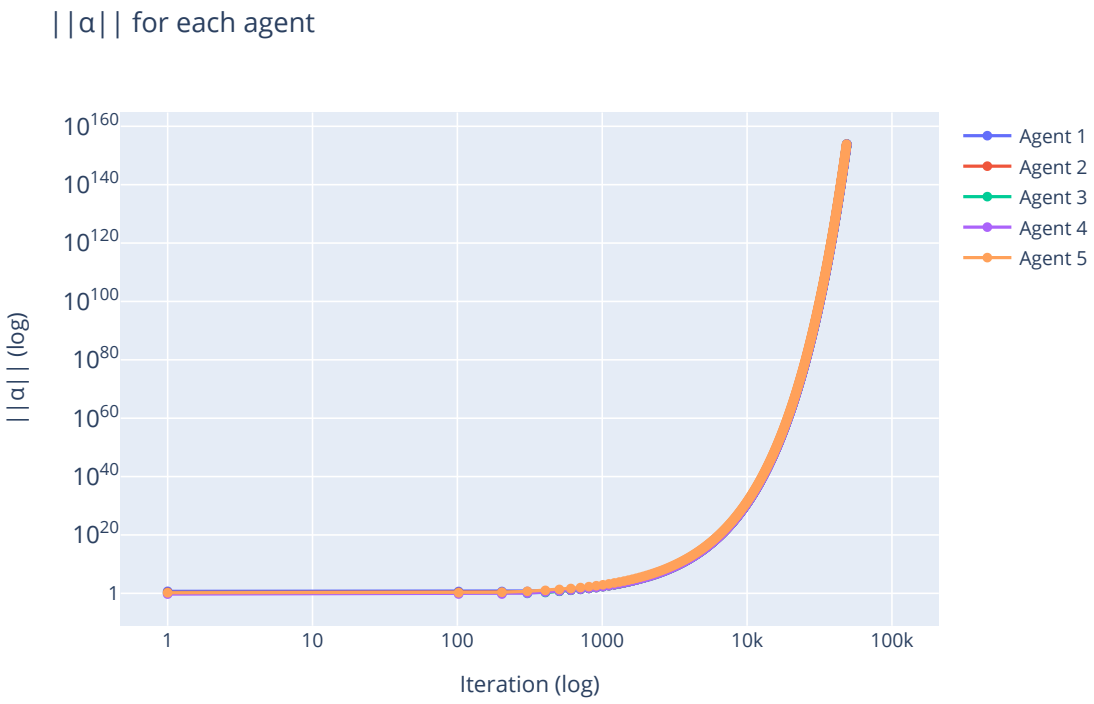


Explosion...

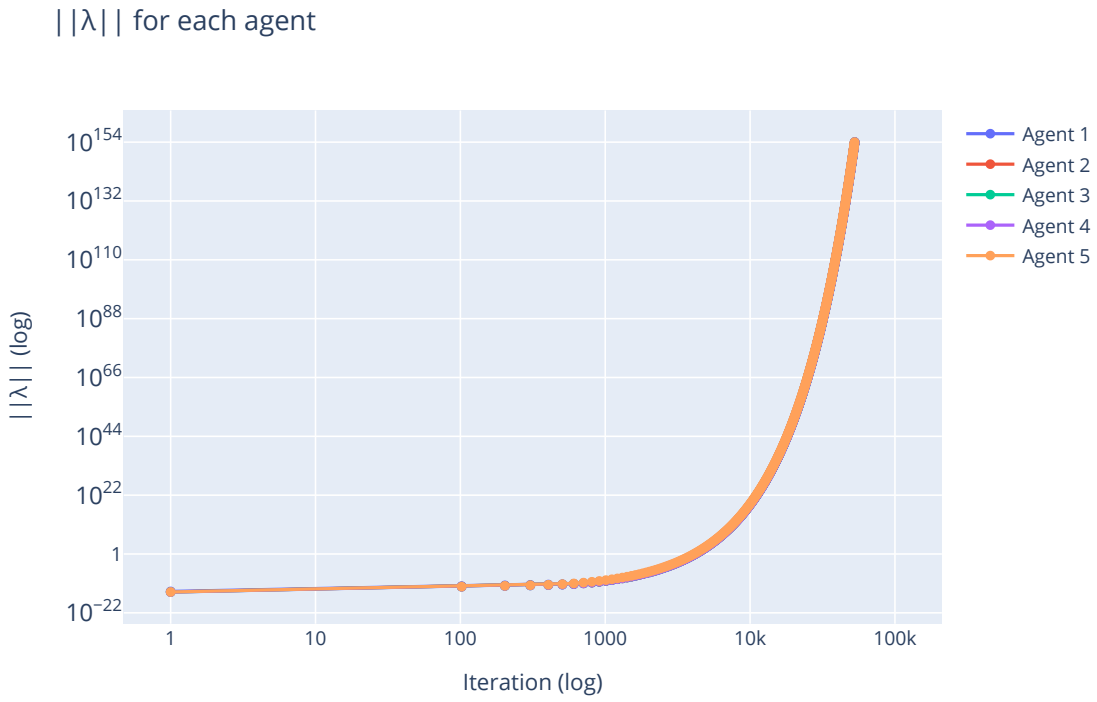
11.9 $||\alpha||$ & $||\lambda||$

[48]:
```
print_vector_norms(a, t_max, alpha_seq, " ")
print_vector_norms(a, t_max, lambda_seq, " ")
```

```
Skipping first iteration for x log plot.
Warning: t_max > 1000, plotting only 1000 points.
```



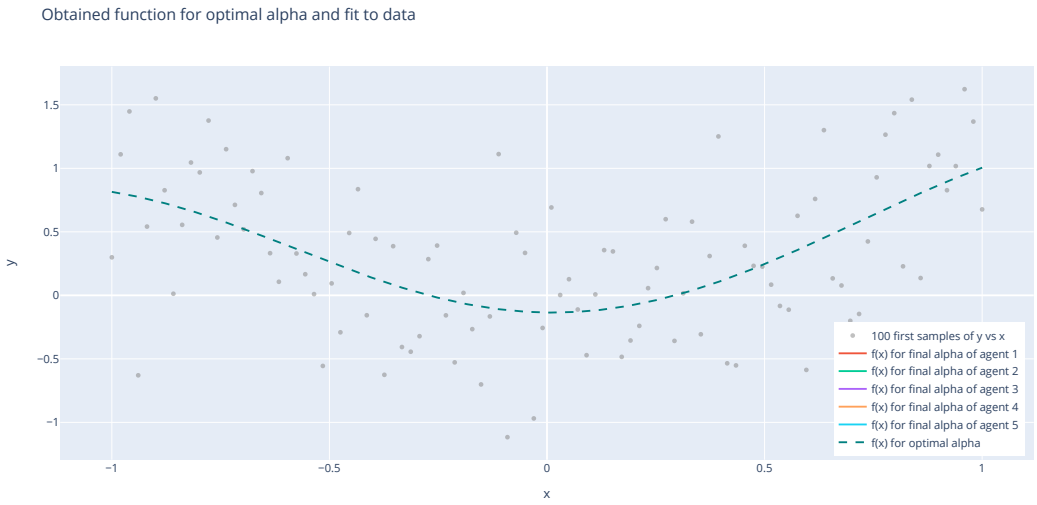

```
Skipping first iteration for x log plot.
Warning: t_max > 1000, plotting only 1000 points.
```



The norm of the variables explodes to infinity...

11.10 Obtained function

[49]:
```
x_prime = np.linspace(-1, 1, 1000)

plot_f(n, x_n, y_n, x_prime, alpha_opti, alpha_seq[-1], x_sel_flat)
```



The Dual Decomposition algorithm never converges, irrespectively of the step size and initialization I choose...

Penalizing the size of the primal variable by adding a $\frac{\beta}{2}||\alpha||_2^2$ term to the objective helps a little bit, but after a long enough time the α's keep diverging.

12 Alternating Direction Method of Multipliers

12.1 Formulation

ADMM introduces a supplementary variable z and solves the following problem:

$$\underset{\substack{\alpha_k, z_{kp} \in \mathbb{R}^m \\ 1 \leq k \leq a \\ p \sim k}}{\arg\min} \sum_{k=1}^{a} f_k(\alpha_k) \quad \text{subject to} \quad \alpha_k = z_{kp} \quad \text{if} \quad W_{kp} \neq 0$$

which can be compactified as:

$$\underset{\alpha \in \mathbb{R}^{am},\, z \in \mathbb{R}^{a^2 m}}{\arg\min} \quad F(\alpha) \quad \text{subject to} \quad A\alpha + Bz = 0$$

One can reformulate this problem as a penalized problem:

$$\alpha_k \leftarrow \underset{\alpha_k \in \mathbb{R}^m}{\arg\min} f_k(\alpha_k) + \frac{\beta}{2} \sum_{p \sim k} ||\alpha_k - z_{kp}||^2$$

the gradient is hence:

$$\nabla f_k(\alpha_k) + \beta \sum_{p \sim k} (\alpha_k - z_{kp})$$

12.2 Convergence conditions

ADMM theoretically converges for *any* step size if: - the f_k's are strongly convex and Lipschitz-smooth, which still holds. - A is full row-rank, which is *not* the case for the default W.

12.3 Hyperparameters

[50]:
```
t_max = int(50e3)  # number of gradient iterations
step_size = 1e-1
```

12.4 Initialization

This time an additional initial *primal* variable z is needed for each agent. I will initialize it randomly, like for α.

[51]:
```
alpha_agents_0 = 2 * rng.random((a, m)) - 1
z_agents_0 = 2 * rng.random((a, a, m)) - 1
```

12.5 ADMM run

[52]:
```
alpha_seq, z_seq = run_ADMM(
    alpha_agents_0,
    z_agents_0,
    t_max,
    a,
    K_mm,
    y_n,
    idx_sel,
    K_nm,
    sigma,
    m,
    W,
    step_size,
)
```

```
  0%|          | 0/50000 [00:00<?, ?it/s]
```

12.6 Optimality gap

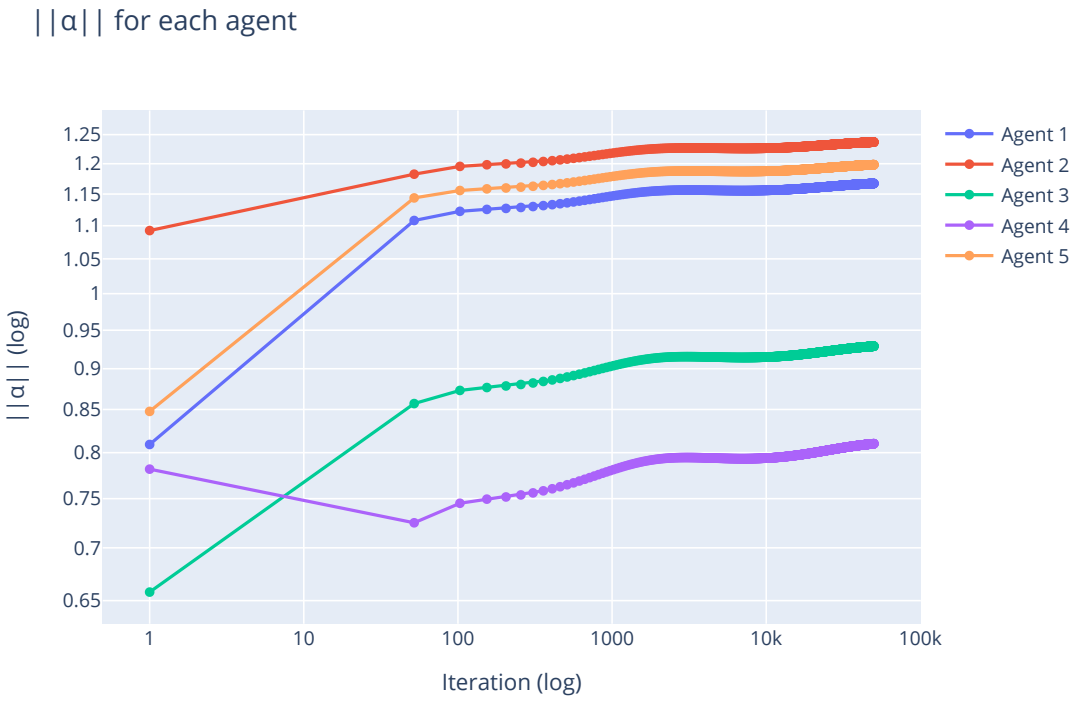
[53]:
```
plot_opt_gap_per_agent(a, t_max, alpha_opti, alpha_seq)
```

```
Warning: t_max > 1000, plotting only 1000 points.
```

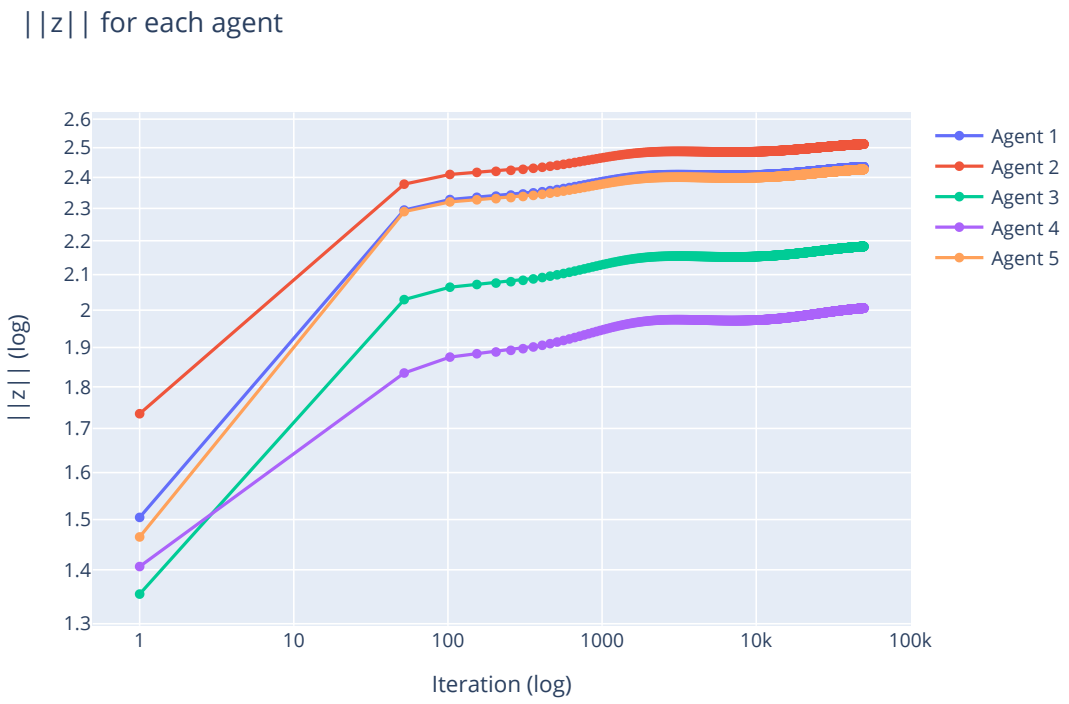
12.7 $||\alpha||$ & $||z||$

[54]:
```
print_vector_norms(a, t_max, alpha_seq, " ")
print_vector_norms(a, t_max, z_seq, "z")
```

```
Skipping first iteration for x log plot.
Warning: t_max > 1000, plotting only 1000 points.
```

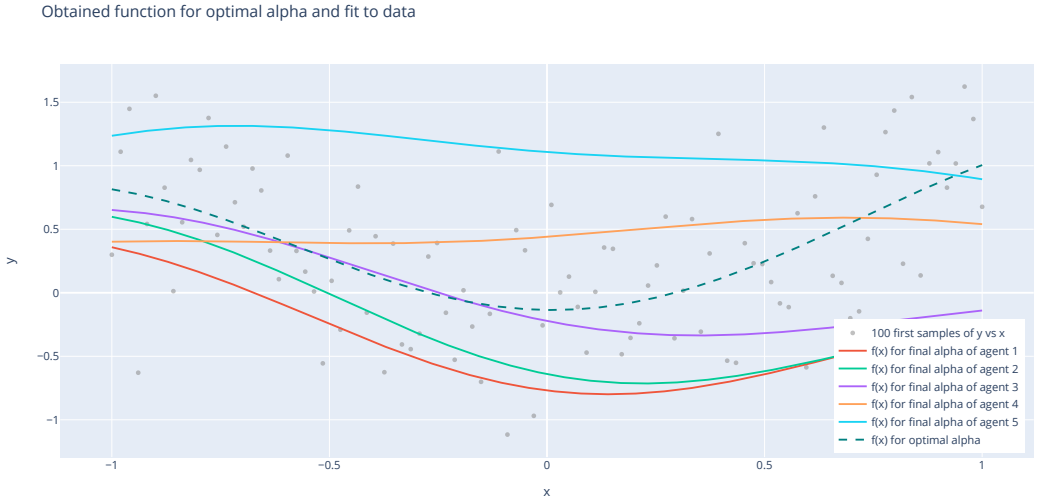



```
Skipping first iteration for x log plot.
Warning: t_max > 1000, plotting only 1000 points.
```



12.8 Obtained function

[55]:
```
x_prime = np.linspace(-1, 1, 1000)

plot_f(n, x_n, y_n, x_prime, alpha_opti, alpha_seq[-1], x_sel_flat)
```



13 Second database

Let's now turn to federative learning.

13.1 Load data

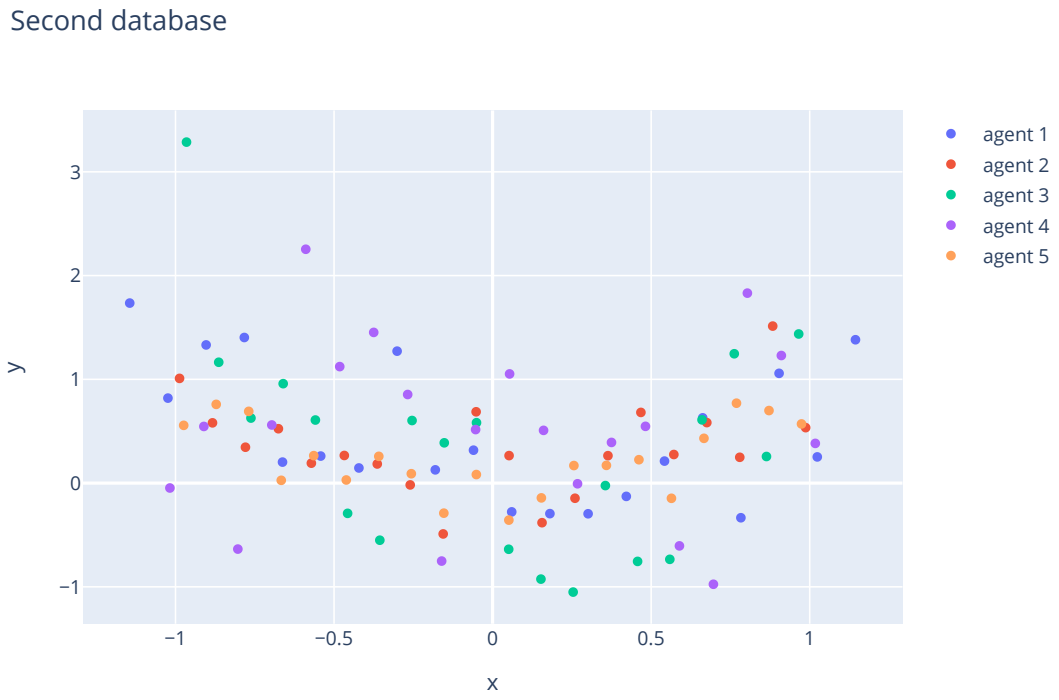
We will use a new databse.

[56]:
```
with open("second_database.pkl", "rb") as f:
    x, y = pickle.load(f)

# here x and y are already splitted among agents
assert len(x) == len(y)
a = len(x)

x = np.array(x)
y = np.array(y)
```

13.2 Visualize data

Let's plot y against x:

[57]:
```
fig = go.Figure()
for k in range(a):
    fig.add_trace(go.Scatter(x=x[k], y=y[k], mode="markers", name=f"agent␣
 ↪{k+1}"))
fig.update_layout(
    title="Second database",
    xaxis_title="x",
    yaxis_title="y",
)
fig.show()
```



13.3 σ

[58]:
```
sigma = 0.5
```

13.4 Kernel matrix

Let's rebuild the kernel matrix:

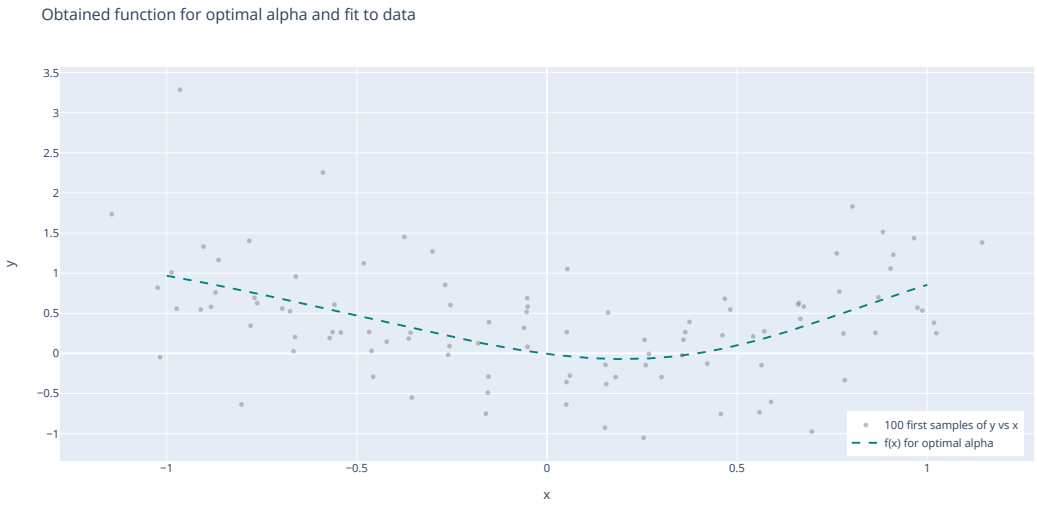
[59]:
```
n = 100
m = 10
x_m_points = np.linspace(-1, 1, m)

K_nm, K_mm, K_dm_per_agent = build_kernel_matrices_out_of_dataset(x_m_points,␣
↪x, n, a)
```

13.5 Optimal α & function

[60]:
```
y_flat = y.flatten()
alpha_opti = np.linalg.solve(
    K_mm + 1 / sigma**2 * K_nm.T @ K_nm, 1 / sigma**2 * K_nm.T @ y_flat
)
print("Optimal alpha:", alpha_opti)
```

```
Optimal alpha: [  8.9794176  -20.76301606  29.53075785 -39.47438522  44.11937528
 -30.26050548  13.00476608  -9.36802235   5.41850291   0.70333259]
```

[61]:
```
x_prime = np.linspace(-1, 1, 1000)

plot_f(n, x.flatten(), y_flat, x_prime, alpha_opti, None, x_m_points)
```



14 FedAvg

14.1 Formulation

The FedAvg algorithm follows the following steps: - each agent k performs a local SGD:

$$\alpha_k \leftarrow \alpha_k - \eta \cdot \nabla f_k(\alpha_k)$$

on some batches of size B_k of its own local data composed of N_k points in total. - the server receives the clients' updates and computes the weighted average:

$$\alpha \leftarrow \sum_{k \in C_t} \frac{N_k}{\sum_{p \in C_t} N_p} \alpha_k$$

where C_t is the set of clients that have sent their updates on this round.

14.2 Convergence conditions

FedAvg converges under the following assumptions: 1. the data is IID among agents 2. the f_k are Lipschitz-smooth 3. the variance of the gradient of the global objective is bounded 4. the step size is diminishing

Condition 1. is hard to assess on such a small dataset, but 2. and 3. are clearly satisfied, and we have control over 4.

14.3 Hyperparameters

Here all the clients will share the same batch size $B_k = B$, and they actually share the same total number of points $N_k = N$ too.

$\frac{N_k}{\sum_{p \in C_t} N_p}$ will thus simply be $\frac{1}{c_t}$ for all clients k, with $c_t = |C_t|$.

Furthermore $c_t = c$ will be constant, and C_t will be randomly chosen at each round.

[62]:
```
B = 10  # batch size for all clients
E = 10  # number of epochs of local SGD
c = 3  # number of clients to be selected at each round

t_max = int(50e3)  # number of FedAvg rounds

# the step size has to diminish with time (inside the local SGD)
# in order for FedAvg to converge; here I choose a linear decrease
start_step_size = 1e-3
end_step_size = 1e-5
slope = (end_step_size - start_step_size) / (E - 1)
step_size_seq = [start_step_size + slope * t for t in range(E)]
```

14.4 Initialization

[63]:
```
alpha_server_0 = 2 * rng.random(m) - 1
```

14.5 FedAvg run

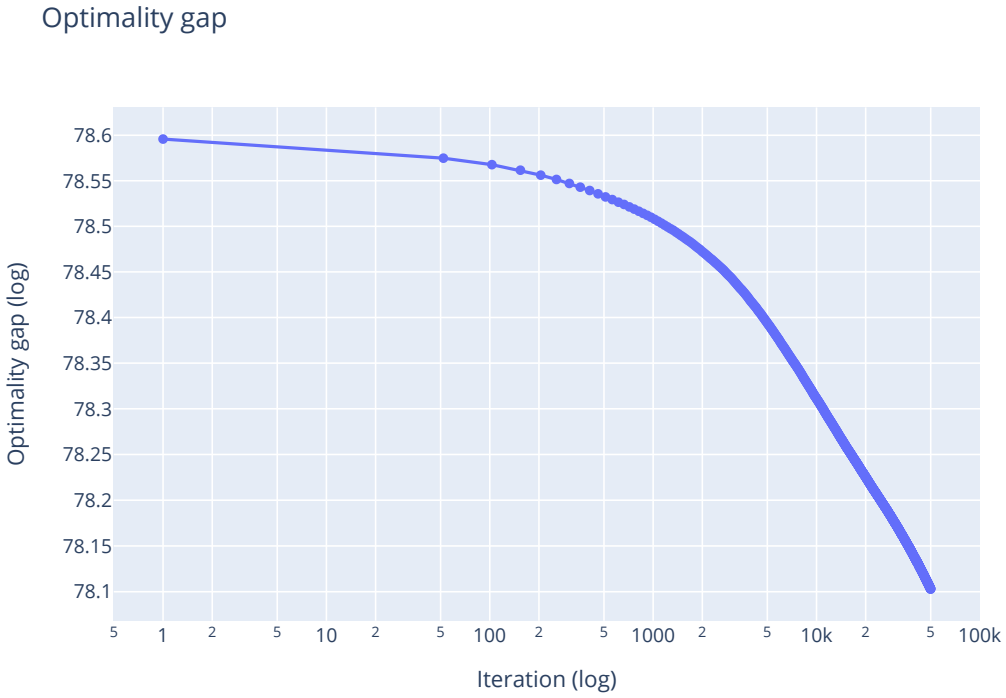
[64]:
```
alpha_seq = run_FedAvg(
    alpha_server_0,
    t_max,
    rng,
    a,
    c,
    m,
    K_dm_per_agent,
    K_mm,
    E,
    x,
    B,
    y,
    sigma,
    step_size_seq,
)
```

```
  0%|          | 0/50000 [00:00<?, ?it/s]
```

14.6 Optimality gap

[65]:
```
plot_opt_gap(t_max, alpha_opti, alpha_seq)
```

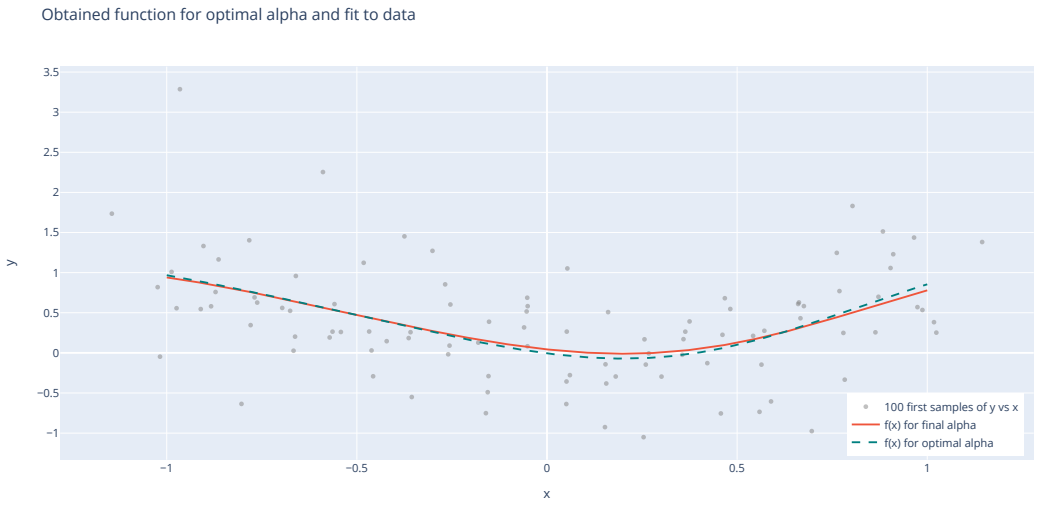
```
Warning: t_max > 1000, plotting only 1000 points.
```



As expected, we seem to have convergence, although it is not really the goal of FedAvg.

14.7 Obtained function

[66]:
```
plot_f(
    n,
    x.flatten(),
    y_flat,
    x_prime,
    alpha_opti,
    alpha_seq[-1].reshape((1, m)),
    x_m_points,
)
```

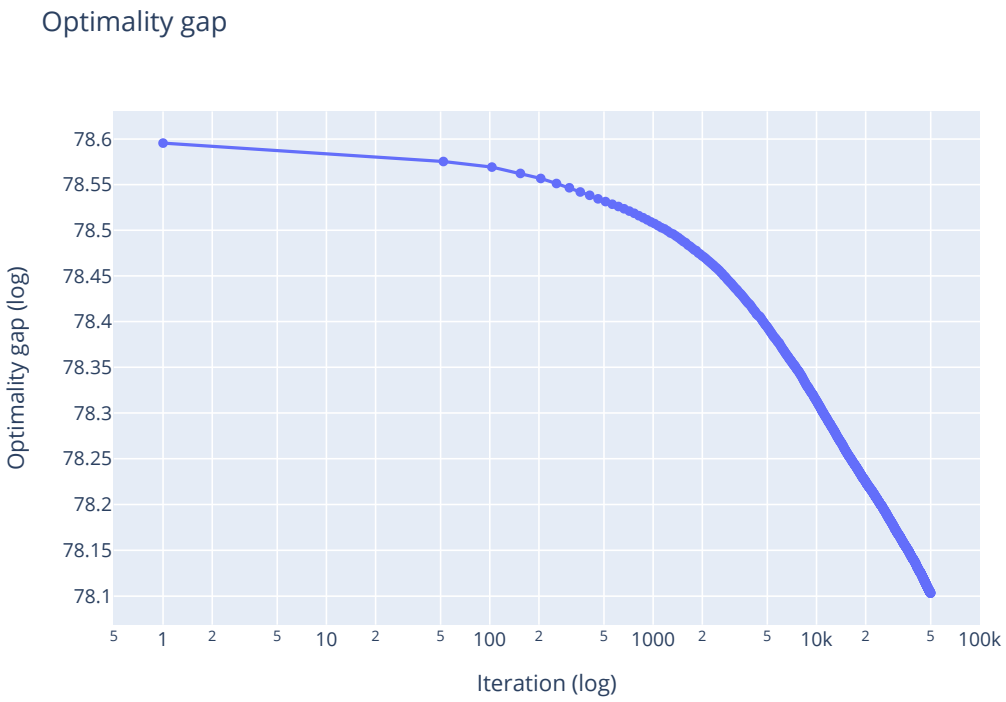


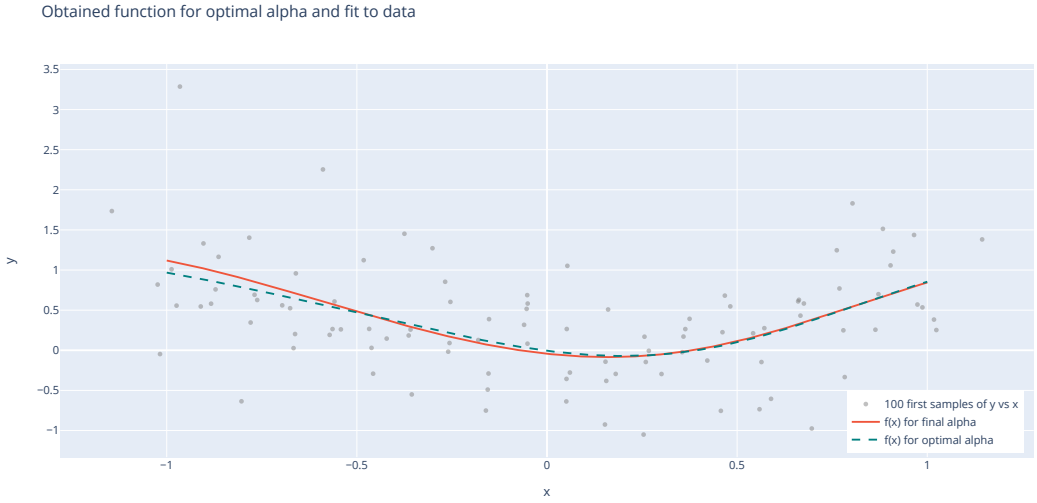
14.8 Variations

Using only one client does not break convergence, as we are essentially performing SGD over the whole dataset:

[67]:
```
# params
c = 1  # number of clients to be selected at each round

study_FedAvg(
    alpha_server_0,
    t_max,
    rng,
    a,
    c,
    m,
    K_dm_per_agent,
    K_mm,
    E,
    x,
    B,
    y,
    sigma,
    step_size_seq,
    alpha_opti,
    n,
    y_flat,
    x_prime,
    x_m_points,
)
```

```
  0%|          | 0/50000 [00:00<?, ?it/s]
Warning: t_max > 1000, plotting only 1000 points.
```

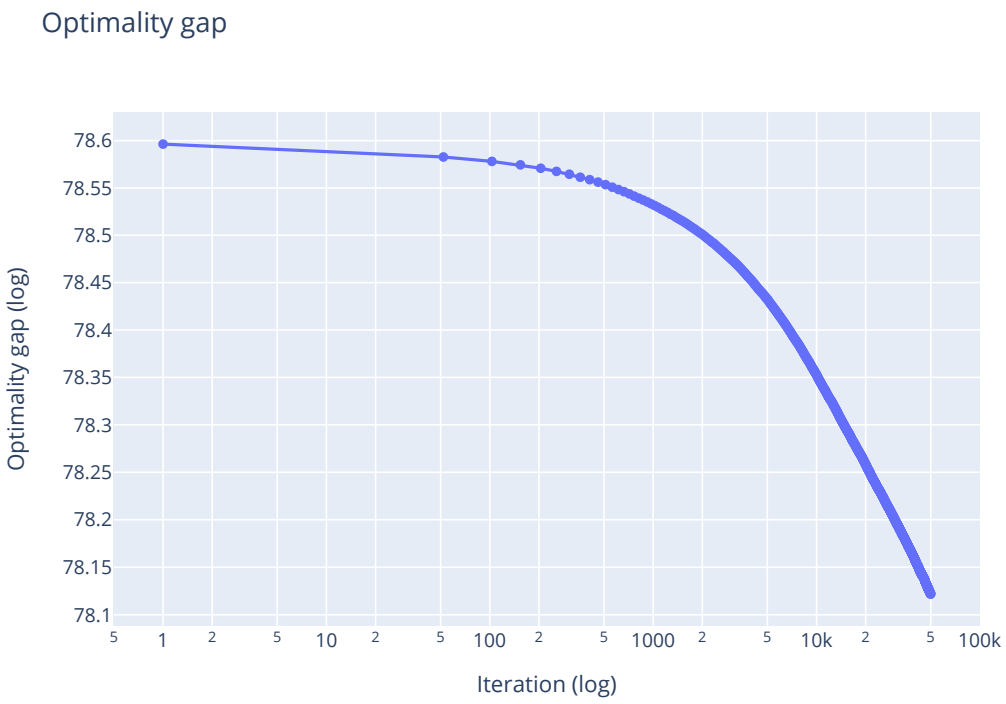


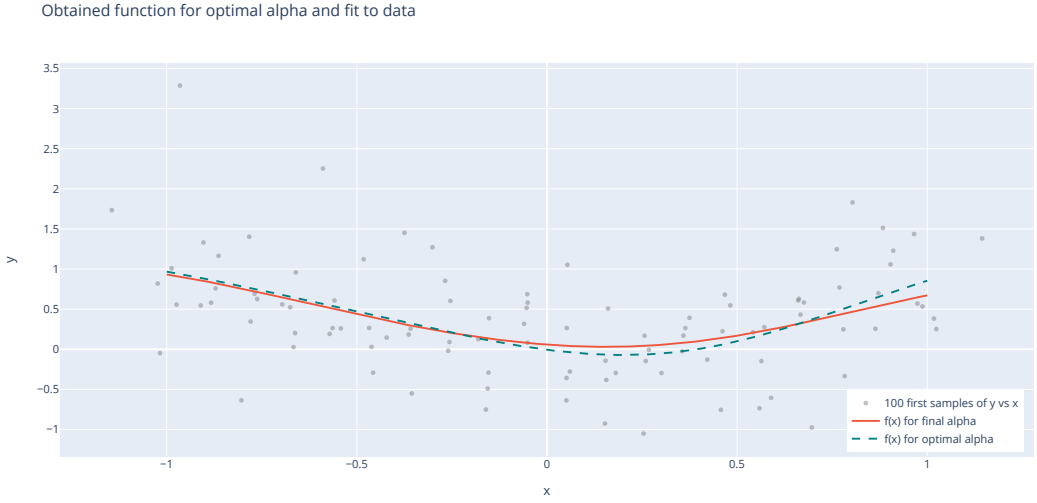


FedAvg seems quite robust:

[68]:
```
# params
c = 2  # number of clients to be selected at each round
E = 1  # number of epochs of local SGD
B = 2  # batch size for all clients

study_FedAvg(
    alpha_server_0,
    t_max,
    rng,
    a,
    c,
    m,
    K_dm_per_agent,
    K_mm,
    E,
    x,
    B,
    y,
    sigma,
    step_size_seq,
    alpha_opti,
    n,
    y_flat,
    x_prime,
    x_m_points,
)
```

```
  0%|          | 0/50000 [00:00<?, ?it/s]
Warning: t_max > 1000, plotting only 1000 points.
```


Optimality gap
Optimality gap (log)
78.6
78.55
78.5
78.45
78.4
78.35
78.3
78.25
78.2
78.15
78.1
5
1
2
5
10
2
5
100
2
5
1000
2
5
10k
2
5
100k
Iteration (log)


Obtained function for optimal alpha and fit to data
y
x
100 first samples of y vs x
f(x) for final alpha
f(x) for optimal alpha